IV Intertwined Data

You might think that the data definitions for lists and natural numbers are quite unusual. These data definitions refer to themselves, and in all likelihood they are the first such definitions you have ever encountered. As it turns out, many classes of data require even more complex data definitions than these two. Common generalizations involve many self-references in one data definition or a bunch of data definitions that refer to each other. These forms of data have become ubiquitous, and it is therefore critical for a programmer to learn to cope with **any** collection of data definitions. And that's what the design recipe is all about.

This part starts with a generalization of the design recipe so that it works for all forms of structural data definitions. Next, it introduces the concept of iterative refinement from Projects: Lists on a rigorous basis because complex data definitions are not developed in one fell swoop but in several stages. Indeed, the use of iterative refinement is one of the reasons why all programmers are little scientists and why our discipline uses the word "science" in its American name. Two last chapters illustrate these ideas: one explains how to design an interpreter for BSL and another is about processing XML, a data exchange language for the web. The last chapter expands the design recipe one more time, reworking it for functions that process two complex arguments at the same time.

19 The Poetry of S-expressions

Programming resembles poetry. Like poets, programmers practice their skill on seemingly pointless ideas. They revise and edit all the time, as the preceding chapter explains. This chapter introduces increasingly complex forms of data—seemingly without a real-world purpose. Even when we provide a motivational background, the chosen kinds of data are pure to an extreme, and it is unlikely that you will ever encounter them again.

Nevertheless, this chapter shows the full power of the design recipe and introduces you to the kinds of data that real-world programs cope with. To connect this material with what you will encounter in your life as a programmer, we label each section with appropriate names: trees, forests, XML. The last one is a bit misleading because it is really about S-expressions; the connection between S-expressions and XML is clarified in Project: The Commerce of XML, which, in contrast to this chapter, comes much closer to real-world uses of complex forms of data.

19.1 Trees

All of us have a family tree. One way to draw a family tree is to add an element every time a child is born and to connect the elements of the father and mother. For those people whose parents are unknown, there is no connection to draw. The result is an *ancestor family tree* because, given any person, the tree points to all of the person's known ancestors.

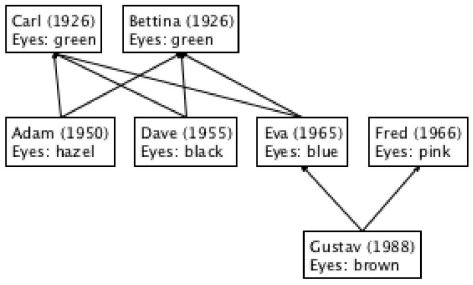


Figure 111: A family tree

Figure 111 displays a three-tier family tree. Gustav is the child of Eva and Fred, while Eva is the child of Carl and Bettina. In addition to people's names and family relationships, the tree also records years of birth and eye colors. Based on this sketch, you can easily imagine a family tree reaching back many generations and one that records other kinds of information.

Once a family tree is large, it makes sense to represent it as data and to design programs that process this kind of data. Given that a point in a family tree combines five pieces of information—the father, the mother, the name, the birth date, and the eye color—we should define a structure type:

```
(define-struct child [father mother name date eyes])
```

The structure type definition calls a data definition:

```
; A Child is a structure:
;   (make-child Child Child String N String)
```

While this data definition looks straightforward, it is also useless. It refers to itself, but, because it doesn't have any clauses, there is no way to create a proper instance Child. Roughly, we would have to write

```
(make-child (make-child (make-child ...) ...) ...)
```

without end. To avoid such pointless data definitions, we demand that a self-referential data definition have several clauses and that at least one of them does not refer back to the data definition.

Let's postpone the data definition for a moment, and experiment instead. Suppose we are about to add a child to an existing family tree and that we already have representations for the parents. In that case, we can simply construct a new `child` structure. For example, to represent Adam in a program that already represents Carl and Bettina, it suffices to add the following `child` structure:

```
(define Adam
  (make-child Carl Bettina "Adam" 1950 "hazel"))
```

assuming `Carl` and `Bettina` stand for representations of Adam's parents.

Then again, a person's parents may be unknown, like Bettina's in the family tree of figure 111. Yet, even then, we must fill the corresponding parent field(s) in the `child` representation. Whatever data we choose, it must signal an absence of information. On the one hand, we could use `#false`, `"none"`, or `'()` from the pool of existing values. On the other hand, we should really say that the information is missing from a family tree. We can achieve this objective best with the introduction of a structure type with an appropriate name:

```
(define-struct no-parent [])
```

Now, to construct a `child` structure for Bettina, we say

```
(make-child (make-no-parent)
            (make-no-parent)
            "Bettina" 1926 "green")
```

Of course, if only one piece of information is missing, we fill just that field with this special value.

Our experimentation suggests two insights. First, we are **not** looking for a data definition that describes how to generate instances of `child` structures but for a data definition that describes how to represent family trees. Second, the data definition consists of two clauses, one for the variant describing unknown family trees and another one for known family trees:

```
(define-struct no-parent [])
(define-struct child [father mother name date eyes])
; An FT (short for family tree) is one of:
; – (make-no-parent)
; – (make-child FT FT String N String)
```

Since the "no parent" tree is going to show up a lot in our programs, we define `NP` as a short-hand and revise the data definition a bit:

```
(define NP (make-no-parent))
; An FT is one of:
; – NP
; – (make-child FT FT String N String)
```

Following the design recipe from Designing with Self-Referential Data Definitions, we use the data definition to create examples of family trees. Specifically, we translate the family tree in figure 111 into our data representation. The information for Carl is easy to translate into data:

```
(make-child NP NP "Carl" 1926 "green")
```

Bettina and Fred are represented with similar instances of `child`. The case for Adam calls for nested children, one for Carl and one for Bettina:

```
(make-child (make-child NP NP "Carl" 1926 "green")
            (make-child NP NP "Bettina" 1926 "green")
            "Adam"
            1950
            "hazel")
```

Since the records for Carl and Bettina are also needed to construct the records for Dave and Eva, it is better to introduce definitions that name specific instances of `child` and to use the variable names elsewhere. Figure 112 illustrates this approach for the complete data representation of the family tree from figure 111. Take a close look; the tree serves as our running example for the following design exercise.

```
; Oldest Generation:
(define Carl (make-child NP NP "Carl" 1926 "green"))
(define Bettina (make-child NP NP "Bettina" 1926 "green"))

; Middle Generation:
(define Adam (make-child Carl Bettina "Adam" 1950 "hazel"))
(define Dave (make-child Carl Bettina "Dave" 1955 "black"))
(define Eva (make-child Carl Bettina "Eva" 1965 "blue"))
(define Fred (make-child NP NP "Fred" 1966 "pink"))

; Youngest Generation:
(define Gustav (make-child Fred Eva "Gustav" 1988 "brown"))
```

Figure 112: A data representation of the sample family tree

Instead of designing a concrete function on family trees, let's first look at the generic organization of such a function. That is, let's work through the design recipe as much as possible without having a concrete task in mind. We start with the header material, that is, step 2 of the recipe:

```
; FT -> ???
; ...
(define (fun-FT an-ftree) ...)
```

Even though we aren't stating the purpose of the function, we do know that it consumes a family tree and that this form of data is the main input. The "???" in the signature says that we don't know what kind of data the function produces; the "..." remind us that we don't know its purpose.

The lack of purpose means we cannot make up functional examples. Nevertheless, we can exploit the organization of the data definition for `FT` to design a template. Since it consists of two clauses, the template must consist of a `cond` expression with two clauses:

```
(define (fun-FT an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else ...]))
```

In case the argument to `fun-FT` satisfies `no-parent?`, the structure contains no additional data, so the first clause is complete. For the second clause, the input contains five pieces of data, which we indicate with five selectors in the template:

```
; FT -> ???
(define (fun-FT an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else (... (child-father an-ftree) ...
           ... (child-mother an-ftree) ...
           ... (child-name an-ftree) ...
           ... (child-date an-ftree) ...
           ... (child-eyes an-ftree) ...)]))
```

The last addition to templates concerns self-references. If a data definition refers to itself, the function is likely to recur and templates indicate so with suggestive natural recursions. The definition for FT has two self-references, and the template therefore needs two such recursions:

```
; FT -> ???
(define (fun-FT an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else (... (fun-FT (child-father an-ftree)) ...
           ... (fun-FT (child-mother an-ftree)) ...
           ... (child-name an-ftree) ...
           ... (child-date an-ftree) ...
           ... (child-eyes an-ftree) ...)]))
```

Specifically, `fun-FT` is applied to the data representation for fathers and mothers in the second `cond` clause because the second clause of the data definition contains corresponding self-references.

Let's now turn to a concrete example, the `blue-eyed-child?` function. Its purpose is to determine whether any `child` structure in a given family tree has blue eyes. You may copy, paste, and rename `fun-FT` to get its template; we replace "???" with Boolean and add a purpose statement:

```
; FT -> Boolean
; does an-ftree contain a child
; structure with "blue" in the eyes field
(define (blue-eyed-child? an-ftree)
  (cond
    [(no-parent? an-ftree) ...]
    [else (... (blue-eyed-child?
                 (child-father an-ftree)) ...
           ... (blue-eyed-child?
                 (child-mother an-ftree)) ...
           ... (child-name an-ftree) ...
           ... (child-date an-ftree) ...
           ... (child-eyes an-ftree) ...)]))
```

When you work in this fashion, you must replace the template's generic name with a specific one.

Checking with our recipe, we realize that we need to backtrack and develop some examples before we move on to the definition step. If we start with Carl, the first person in the family tree, we see that Carl's family tree does not contain a `child` with a `"blue"` eye color. Specifically, the `child` representing Carl says the eye color is `"green"`; given that Carl's ancestor trees are empty, they cannot possibly contain a `child` with `"blue"` eye color:

```
(check-expect (blue-eyed-child? Carl) #false)
```

In contrast, `Gustav` contains a `child` for `Eva` who does have blue eyes:

```
(check-expect (blue-eyed-child? Gustav) #true)
```

Now we are ready to define the actual function. The function distinguishes between two cases: a `no-parent` and a `child`. For the first case, the answer should be obvious even though we haven't made up any examples. Since the given family tree does not contain any `child` whatsoever, it cannot contain one with `"blue"` as the eye color. Hence the result in the first `cond` clause is `#false`.

For the second `cond` clause, the design requires a lot more work. Again following the design recipe, we first remind ourselves what the expressions in the template accomplish:

1. according to the purpose statement for the function,

   ```
   (blue-eyed-child? (child-father an-ftree))
   ```

   determines whether some `child` in the father's FT has `"blue"` eyes;

2. likewise, `(blue-eyed-child? (child-mother an-ftree))` determines whether someone in the mother's FT has blue eyes; and

3. the selector expressions `(child-name an-ftree)`, `(child-date an-ftree)`, and `(child-eyes an-ftree)` extract the name, birth date, and eye color from the given `child` structure, respectively.

Now we just need to figure out how to combine these expressions.

Clearly, if the `child` structure contains `"blue"` in the `eyes` field, the function's answer is `#true`. Next, the expressions concerning names and birth dates are useless, which leaves us with the recursive calls. As stated, `(blue-eyed-child? (child-father an-ftree))` traverses the tree on the father's side, while the mother's side of the family tree is processed with `(blue-eyed-child? (child-mother an-ftree))`. If either of these expressions returns `#true`, `an-ftree` contains a `child` with `"blue"` eyes.

Our analysis suggests that the result should be `#true` if one of the following three expressions is `#true`:

- `(string=? (child-eyes an-ftree) "blue")`
- `(blue-eyed-child? (child-father an-ftree))`
- `(blue-eyed-child? (child-mother an-ftree))`

which, in turn, means we need to combine these expressions with `or`:

```
(or (string=? (child-eyes an-ftree) "blue")
    (blue-eyed-child? (child-father an-ftree))
    (blue-eyed-child? (child-mother an-ftree)))
```

Figure 113 pulls everything together in a single definition.

```
; FT -> Boolean
```

```
; does an-ftree contain a child
; structure with "blue" in the eyes field

(check-expect (blue-eyed-child? Carl) #false)
(check-expect (blue-eyed-child? Gustav) #true)

(define (blue-eyed-child? an-ftree)
  (cond
    [(no-parent? an-ftree) #false]
    [else (or (string=? (child-eyes an-ftree) "blue")
              (blue-eyed-child? (child-father an-ftree))
              (blue-eyed-child? (child-mother an-ftree)))]))
```

Figure 113: Finding a blue-eyed child in an ancestor tree

Since this function is the very first one to use two recursions, we simulate the stepper's action for `(blue-eyed-child? Carl)` to give you an impression of how it all works:

```
(blue-eyed-child? Carl)
==
(blue-eyed-child?
  (make-child NP NP "Carl" 1926 "green"))
```

Let's act as if `NP` were a value and let's use `carl` as an abbreviation for the instance of `child`:

```
==
(cond
  [(no-parent?
     (make-child NP NP "Carl" 1926 "green"))
   #false]
  [else (or (string=? (child-eyes carl) "blue")
            (blue-eyed-child? (child-father carl))
            (blue-eyed-child? (child-mother carl)))])
```

After dropping the first `cond` line, it's time to replace `carl` with its value and to perform the three auxiliary calculations in figure 114. Using these to replace equals with equals, the rest of the computation is explained easily:

```
==
(or (string=? "green" "blue")
    (blue-eyed-child? (child-father carl))
    (blue-eyed-child? (child-mother carl)))
== (or #false #false #false)
== #false
```

While we trust that you have seen such auxiliary calculations in your mathematics courses, you also need to understand that the stepper would **not** perform such calculations; instead it works out only those calculations that are absolutely needed.

```
; (1)
(child-eyes (make-child NP NP "Carl" 1926 "green"))
==
"green"

; (2)
(blue-eyed-child?
  (child-father
    (make-child NP NP "Carl" 1926 "green")))
==
(blue-eyed-child? NP)
==
#false

; (3)
(blue-eyed-child?
  (child-mother
    (make-child NP NP "Carl" 1926 "green")))
==
(blue-eyed-child? NP)
```

```
  ==
  #false
```

Figure 114: Calculating with trees

Exercise 310. Develop `count-persons`. The function consumes a family tree and counts the `child` structures in the tree.

Exercise 311. Develop the function `average-age`. It consumes a family tree and the current year. It produces the average age of all `child` structures in the family tree.

Exercise 312. Develop the function `eye-colors`, which consumes a family tree and produces a list of all eye colors in the tree. An eye color may occur more than once in the resulting list. **Hint** Use `append` to concatenate the lists resulting from the recursive calls.

Exercise 313. Suppose we need the function `blue-eyed-ancestor?`, which is like `blue-eyed-child?` but responds with `#true` only when a proper ancestor, not the given `child` itself, has blue eyes.

Although the goals clearly differ, the signatures are the same:

```
; FT -> Boolean
(define (blue-eyed-ancestor? an-ftree) ...)
```

Stop! Formulate a purpose statement for the function.

To appreciate the difference, we take a look at Eva:

```
(check-expect (blue-eyed-child? Eva) #true)
```

Eva is blue-eyed, but has no blue-eyed ancestor. Hence,

```
(check-expect (blue-eyed-ancestor? Eva) #false)
```

In contrast, Gustav is Eva's son and does have a blue-eyed ancestor:

```
(check-expect (blue-eyed-ancestor? Gustav) #true)
```

Now suppose a friend comes up with this solution:

```
(define (blue-eyed-ancestor? an-ftree)
  (cond
    [(no-parent? an-ftree) #false]
    [else
     (or
       (blue-eyed-ancestor?
         (child-father an-ftree))
       (blue-eyed-ancestor?
         (child-mother an-ftree)))]))
```

Explain why this function fails one of its tests. What is the result of `(blue-eyed-ancestor? A)` no matter which `A` you choose? Can you fix your friend's solution?

19.2 Forests

It is a short step from a family tree to a family forest:

```
; An FF (short for family forest) is one of:
; – '()
; – (cons FT FF)
; interpretation a family forest represents several
; families (say, a town) and their ancestor trees
```

Here are some trees excerpts from figure 111 arranged as forests:

```
(define ff1 (list Carl Bettina))
(define ff2 (list Fred Eva))
(define ff3 (list Fred Eva Carl))
```

The first two forests contain two unrelated families, and the third one illustrates that unlike in real forests, trees in family forests can overlap.

Now consider this representative problem concerning family trees:

Sample Problem Design the function `blue-eyed-child-in-forest?`, which determines whether a family forest contains a `child` with `"blue"` in the `eyes` field.

```
; FF -> Boolean
; does the forest contain any child with "blue" eyes

(check-expect (blue-eyed-child-in-forest? ff1) #false)
(check-expect (blue-eyed-child-in-forest? ff2) #true)
(check-expect (blue-eyed-child-in-forest? ff3) #true)

(define (blue-eyed-child-in-forest? a-forest)
  (cond
    [(empty? a-forest) #false]
    [else
     (or (blue-eyed-child? (first a-forest))
         (blue-eyed-child-in-forest? (rest a-forest)))]))
```

Figure 115: Finding a blue-eyed child in a family forest

The straightforward solution is displayed in figure 115. Study the signature, the purpose statement, and the examples on your own. We focus on the program organization. Concerning the template, the design may employ the list template because the function consumes a list. If each item on the list were a structure with an `eyes` field and nothing else, the function would iterate over those structures using the selector function for the `eyes` field and a string comparison. In this case, each item is a family tree, but, luckily, we already know how to process family trees.

Let's step back and inspect how we explained figure 115. The starting point is a **pair** of data definitions where the second refers to the first and both refer to themselves. The result is a **pair** of functions where the second refers to the first and both refer to themselves. In other words, the function definitions refer to each other the same way the data definitions refer to each other. Early chapters gloss over this kind of relationship, but now the situation is sufficiently complicated and deserves attention.

Exercise 314. Reformulate the data definition for FF with the List-of abstraction. Now do the same for the `blue-eyed-child-in-forest?` function. Finally, define `blue-eyed-child-in-forest?` using one of the list abstractions from the preceding chapter.

Exercise 315. Design the function `average-age`. It consumes a family forest and a year (N). From this data, it produces the average age of all `child` instances in the forest. **Note** If the trees in this forest overlap, the result isn't a true average because some people contribute more than others. For this exercise, act as if the trees don't overlap.

19.3 S-expressions

While Intermezzo 2: Quote, Unquote introduced S-expressions on an informal basis, it is possible to describe them with a combination of three data definitions:

```
; An S-expr is one of:          ; An Atom is one of:
; – Atom                       ; – Number
; – SL                         ; – String
                               ; – Symbol
; An SL is one of:
; – '()
; – (cons S-expr SL)
```

Recall that Symbols look like strings with a single quote at the beginning and with no quote at the end.

The idea of S-expressions is due to John McCarthy and his Lispers, who created S-expressions in 1958 so that they could process Lisp programs with other Lisp programs. This seemingly circular reasoning may sound esoteric, but, as mentioned in Intermezzo 2: Quote, Unquote, S-expressions are a versatile form of data that is often rediscovered, most recently with applications to the world wide web. Working with S-expressions thus prepares a discussion of how to design functions for highly intertwined data definitions.

Exercise 316. Define the `atom?` function.

Up to this point in this book, no data has required a data definition as complex as the one for S-expressions. And yet, with one extra hint, you can design functions that process S-expressions if you follow the design recipe. To demonstrate this point, let's work through a specific example:

Sample Problem Design the function `count`, which determines how many times some symbol occurs in some S-expression.

While the first step calls for data definitions and appears to have been completed, remember that it also calls for the creation of data examples, especially when the definition is complex.

A data definition is supposed to be a prescription of how to create data, and its "test" is whether it is usable. One point that the data definition for S-expr makes is that every Atom is an element of S-expr, and you know that Atoms are easy to fabricate:

```
'hello
20.12
"world"
```

In the same vein, every SL is a list as well as an S-expr:

```
'()
(cons 'hello (cons 20.12 (cons "world" '())))
(cons (cons 'hello (cons 20.12 (cons "world" '())))
      '())
```

The first two are obvious; the third one deserves a second look. It repeats the second S-expr but nested inside `(cons ... '())`. What this means is that it is a list that contains a single item, namely, the second example. You can simplify the example with `list`:

```
(list (cons 'hello (cons 20.12 (cons "world" '()))))
; or
(list (list 'hello 20.12 "world"))
```

Indeed, with the quotation mechanism of Intermezzo 2: Quote, Unquote it is even easier to write down S-expressions. Here are the last three:

```
> '()
'()
> '(hello 20.12 "world")
(list 'hello #i20.12 "world")
> '((hello 20.12 "world"))
(list (list 'hello #i20.12 "world"))
```

To help you out, we evaluate these examples in the interactions area of DrRacket so that you can see the result, which is closer to the above constructions than the `quote` notation.

With `quote`, it is quite easy to make up complex examples:

```
> '(define (f x)
     (+ x 55))
(list 'define (list 'f 'x) (list '+ 'x 55))
```

This example may strike you as odd because it looks like a definition in BSL, but, as the interaction with DrRacket shows, it is just a piece of data. Here is another one:

```
> '((6 f)
    (5 e)
    (4 d))
(list (list 6 'f) (list 5 'e) (list 4 'd))
```

This piece of data looks like a table, associating letters with numbers. The last example is a piece of art:

```
> '(wing (wing body wing) wing)
(list 'wing (list 'wing 'body 'wing) 'wing)
```

It is now time to write down the rather obvious header for `count`:

```
; S-expr Symbol -> N
; counts all occurrences of sy in sexp
(define (count sexp sy)
  0)
```

Since the header is obvious, we move on to functional examples. If the given S-expr is `'world` and the to-be-counted symbol is `'world`, the answer is obviously `1`. Here are some more examples, immediately formulated as tests:

```
(check-expect (count 'world 'hello) 0)
(check-expect (count '(world hello) 'hello) 1)
(check-expect (count '(((world) hello) hello) 'hello) 2)
```

You can see how convenient quotation notation is for test cases. When it comes to templates, however, thinking in terms of `quote` is disastrous.

Before we move on to the template step, we need to prepare you for the next generalization of the design recipe:

> **Hint** For intertwined data definitions, create one template per data definition. Create them in parallel. Make sure they refer to each other in the same way the data definitions do. **End**

This hint sounds more complicated than it is. For our problem, it means we need three templates:

1. one for `count`, which counts occurrences of symbols in S-exprs;
2. one for a function that counts occurrences of symbols in SLs; and
3. one for a function that counts occurrences of symbols in Atoms.

And here are three partial templates, with conditionals as suggested by the three data definitions:

```
(define (count sexp sy)
 (cond
   [(atom? sexp) ...]
   [else ...]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else ...]))
```

```
(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...]))
```

The template for `count` contains a two-pronged conditional because the data definition for S-expr has two clauses. It uses the `atom?` function to distinguish the case for Atoms from the case for SLs. The template named `count-sl` consumes an element of SL and a symbol, and because SL is basically a list, `count-sl` also contains a two-pronged `cond`. Finally, `count-atom` is supposed to work for both Atoms and Symbols. And this means that its template checks for the three distinct forms of data mentioned in the data definition of Atom.

The next step is to take apart compound data in the relevant clauses:

```
(define (count sexp sy)
 (cond
   [(atom? sexp) ...]
   [else ...]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else
     (... (first sl) ...
      ... (rest sl))]))
```

```
(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...]))
```

Why do we add just two selector expressions to `count-sl`?

The last step in the template creation process calls for an inspection of self-references in the data definitions. In our context, this means self-references **and** references from one data definition to another and (possibly) back. Let's inspect the `cond` lines in the three templates:

1. The `atom?` line in `count` corresponds to the first line in the definition of S-expr. To indicate the cross-reference from here to Atom, we add `(count-atom sexp sy)`, meaning we interpret `sexp` as an Atom and let the appropriate function deal with it.
2. Following the same line of thought, the second `cond` line in `count` calls for the addition of `(count-sl sexp sy)`.
3. The `empty?` line in `count-sl` corresponds to a line in the data definition that makes no reference to another data definition.
4. In contrast, the `else` line contains two selector expressions, and each extracts a different kind of value. Specifically, `(first sl)` is an element of S-expr, which means that we wrap it in `(count ...)`. After all, `count` is responsible for counting inside of arbitrary S-exprs. Next, `(rest sl)` corresponds to a self-reference, and we know that we need to deal with those via recursive function calls.

5. Finally, all three cases in Atom refer to atomic forms of data. Therefore the `count-atom` function does not need to change.

```
(define (count sexp sy)
 (cond
   [(atom? sexp)
    (count-atom sexp sy)]
   [else
    (count-sl sexp sy)]))

(define (count-sl sl sy)
  (cond
    [(empty? sl) ...]
    [else
     (...
      (count (first sl) sy)
      ...
      (count-sl (rest sl) sy)
      ...)]))
```

```
(define (count-atom at sy)
  (cond
    [(number? at) ...]
    [(string? at) ...]
    [(symbol? at) ...]))
```

Figure 116: A template for S-expressions

```
; S-expr Symbol -> N
; counts all occurrences of sy in sexp
(define (count sexp sy)
 (cond
   [(atom? sexp) (count-atom sexp sy)]
   [else (count-sl sexp sy)]))

; SL Symbol -> N
; counts all occurrences of sy in sl
(define (count-sl sl sy)
  (cond
    [(empty? sl) 0]
    [else
     (+ (count (first sl) sy) (count-sl (rest sl) sy))]))

; Atom Symbol -> N
; counts all occurrences of sy in at
(define (count-atom at sy)
  (cond
    [(number? at) 0]
    [(string? at) 0]
    [(symbol? at) (if (symbol=? at sy) 1 0)]))
```

Figure 117: A program for S-expressions

Figure 116 presents the three complete templates. Filling in the blanks in these templates is straightforward, as figure 117 shows. You ought to be able to explain any random line in the three definitions. For example:

```
[(atom? sexp) (count-atom sexp sy)]
```

determines whether `sexp` is an Atom and, if so, interprets the S-expr as an Atom via `count-atom`.

```
[else
 (+ (count (first sl) sy) (count-sl (rest sl) sy))]
```

means the given list consists of two parts: an S-expr and an SL. By using `count` and `count-sl`, the corresponding functions are used to count how often `sy` appears in each part, and the two numbers are added up—yielding the total number of `sy`s in all of `sexp`.

```
[(symbol? at) (if (symbol=? at sy) 1 0)]
```

tells us that if an Atom is a Symbol, `sy` occurs once if it is equal to `sexp` and otherwise it does not occur at all. Since the two pieces of data are atomic, there is no other possibility.

Exercise 317. A program that consists of three connected functions ought to express this relationship with a `local` expression.

Copy and reorganize the program from figure 117 into a single function using `local`. Validate the revised code with the test suite for `count`.

The second argument to the local functions, `sy`, never changes. It is always the same as the original symbol. Hence you can eliminate it from the local function definitions to tell the reader that `sy` is a constant across the traversal process.

Exercise 318. Design `depth`. The function consumes an S-expression and determines its depth. An Atom has a depth of `1`. The depth of a list of S-expressions is the maximum depth of its items plus `1`.

Exercise 319. Design `substitute`. It consumes an S-expression `s` and two symbols, `old` and `new`. The result is like `s` with all occurrences of `old` replaced by `new`.

Exercise 320. Reformulate the data definition for S-expr so that the first clause is expanded into the three clauses of Atom and the second clause uses the List-of abstraction. Redesign the `count` function for this data definition.

Now integrate the definition of SL into the one for S-expr. Simplify `count` again. Consider using `lambda`.

Note This kind of simplification is not always possible, but experienced programmers tend to recognize such opportunities.

Exercise 321. Abstract the data definitions for S-expr and SL so that they abstract over the kinds of Atoms that may appear.

19.4 Designing with Intertwined Data

The jump from self-referential data definitions to collections of data definitions with mutual references is far smaller than the one from data definitions for finite data to self-referential data definitions. Indeed, the design recipe for self-referential data definitions—see Designing with Self-Referential Data Definitions—needs only minor adjustments to apply to this seemingly complex situation:

1. The need for “nests” of mutually related data definitions is similar to the one for the need for self-referential data definitions. The problem statement deals with many distinct kinds of information, and one form of information refers to other kinds.

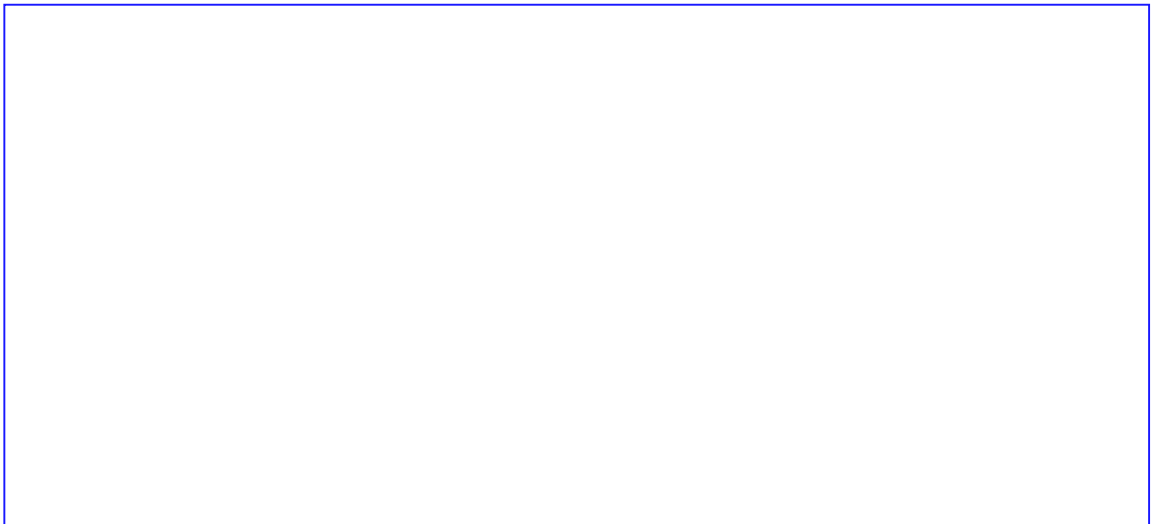
   Before you proceed in such situations, draw arrows to connect references to definitions. Consider the left side of figure 118. It displays the definition for S-expr, which contains references to SL and Atom that are connected to their respective definitions via arrows. Similarly, the definition of SL contains one self-reference and one reference back to SL; again, both are connected by appropriate arrows.

   Like self-referential data definitions, these nests of definitions also call for validation. At a minimum, you must be able to construct some examples for every individual definition. Start from clauses that do not refer to any of the other data definitions in the nest. Keep in mind that the definition may be invalid if it is impossible to generate examples from them.

2. The key change is that you must design as many functions in parallel as there are data definitions. Each function specializes for one of the data definitions; all remaining arguments remain the same. Based on that, you start with a signature, a purpose statement, and a dummy definition **for each function**.

3. Be sure to work through functional examples that use all mutual references in the nest of data definitions.

4. For each function, design the template according to its primary data definition. Use figure 52 to guide the template creation up to the last step. The revised last step calls for a check for all self-references and cross-references. Use the data definitions annotated with arrows to guide this step. For each arrow in the data definitions, include an arrow in the templates. See the right side of figure 118 for the arrow-annotated version of the templates.



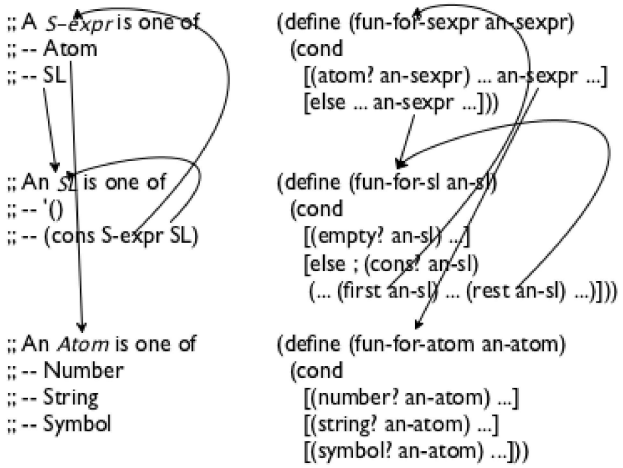


Figure 118: Arrows for nests of data definitions and templates

Now replace the arrows with actual function calls. As you gain experience, you will naturally skip the arrow-drawing step and use function calls directly.

Note Observe how both nests—the one for data definitions and the one for function templates—contain four arrows, and note how pairs of arrows correspond to each other. Researchers call this correspondence a *symmetry*. It is evidence that the design recipe provides a natural way for going from problems to solutions.

5. For the design of the body, we start with those `cond` lines that do not contain natural recursions or calls to other functions. They are called *base cases*. The corresponding answers are typically easy to formulate or are already given by the examples. After that, you deal with the self-referential cases and the cases of cross-function calls. Let the questions and answers of figure 53 guide you.

6. Run the tests when all definitions are completed. If an auxiliary function is broken, you may get two error reports, one for the main function and another one for the flawed auxiliary definition. A **single** fix should eliminate both. Do make sure that running the tests covers all the pieces of the function.

Finally, if you are stuck in step 5, remember the table-based approach to guessing the combination function. In the case of intertwined data, you may need not only a table per case but also a table per case and per function to work out the combination.

19.5 Project: BSTs

Programmers often work with tree representations of data to improve the performance of their functions. A particularly well-known form of tree is the **binary search tree** because it is a good way to store and retrieve information quickly.

To be concrete, let's discuss binary trees that manage information about people. Instead of the `child` structures in family trees, a binary tree contains `node`s:

```
(define-struct no-info [])
(define NONE (make-no-info))

(define-struct node [ssn name left right])
; A BT (short for BinaryTree) is one of:
; – NONE
; – (make-node Number Symbol BT BT)
```

The corresponding data definition is like the one for family trees with `NONE` indicating a lack of information and each `node` recording a social security number, a name, and two other binary trees. The latter are like the parents of family trees, though the relationship between a `node` and its `left` and `right` trees is not based on family relationships.

Here are two binary trees:

```
(make-node
  15
  'd
  NONE
  (make-node
    24 'i NONE NONE))
```

```
(make-node
  15
  'd
  (make-node
    87 'h NONE NONE)
  NONE)
```

Figure 119 shows how we should think about such trees as drawings. The trees are drawn upside down, with the root at the top and the crown of the tree at the bottom. Each circle corresponds to a node, labeled with the `ssn` field of a

corresponding `node` structure. The drawings omit `NONE`.

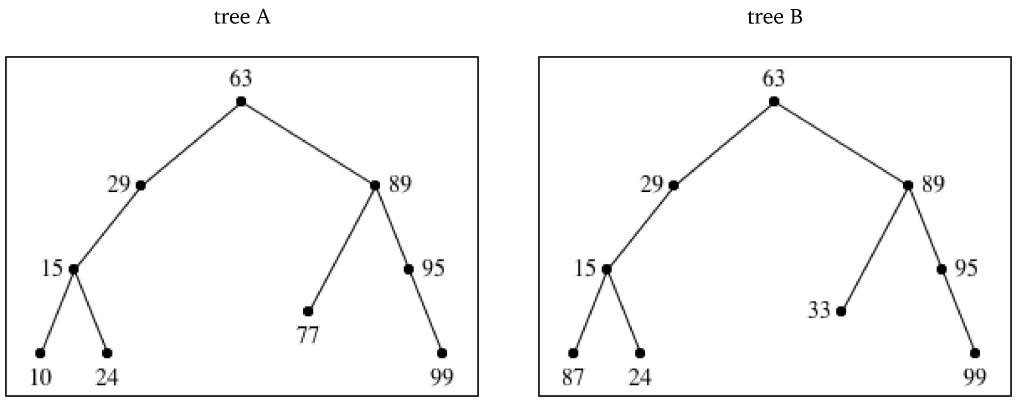


Figure 119: A binary search tree and a binary tree

Exercise 322. Draw the above two trees in the manner of figure 119. Then design `contains-bt?`, which determines whether a given number occurs in some given `BT`.

Exercise 323. Design `search-bt`. The function consumes a number `n` and a `BT`. If the tree contains a `node` structure whose `ssn` field is `n`, the function produces the value of the `name` field in that node. Otherwise, the function produces `#false`.

Hint Consider using `contains-bt?` to check the entire tree first or `boolean?` to check the result of the natural recursion at each stage.

If we read the numbers in the two trees in figure 119 from left to right, we obtain two different sequences:

tree A	10	15	24	29	63	77	89	95	99
tree B	87	15	24	29	63	33	89	95	99

The sequence for tree A is sorted in ascending order the one for B is not. A binary tree of the first kind is a **binary search tree**. Every binary search tree is a binary tree, but not every binary tree is a binary search tree. More concretely, we formulate a condition—or data invariant—that distinguishes a binary search tree from a binary tree:

> *The BST Invariant*
>
> A *BST* (short for *binary search tree*) is a BT according to the following conditions:
>
> - `NONE` is always a BST.
> - `(make-node ssn0 name0 L R)` is a BST if
>   - `L` is a BST,
>   - `R` is a BST,
>   - all `ssn` fields in `L` are smaller than `ssn0`,
>   - all `ssn` fields in `R` are larger than `ssn0`.

In other words, to check whether a BT also belongs to BST, we must inspect all numbers in all subtrees and ensure that they are smaller or larger than some given number. This places an additional burden on the construction of data, but, as the following exercises show, it is well worth it.

Exercise 324. Design the function `inorder`. It consumes a binary tree and produces the sequence of all the `ssn` numbers in the tree as they show up from left to right when looking at a tree drawing.

Hint Use `append`, which concatenates lists like thus:

```
(append (list 1 2 3) (list 4) (list 5 6 7))
==
(list 1 2 3 4 5 6 7)
```

What does `inorder` produce for a binary search tree?

Looking for a `node` with a given `ssn` in a `BST` may exploit the BST invariant. To find out whether a `BT` contains a node with a specific `ssn`, a function may have to look at every `node` of the tree. In contrast, to find out whether a binary **search** tree contains the same `ssn`, a function may eliminate one of two subtrees for every node it inspects.

Let's illustrate the idea with this sample `BST`:

```
(make-node 66 'a L R)
```

If we are looking for 66, we have found the node we are looking for. Now, if we are looking for a smaller number, say 63, we can focus the search on L because **all** nodes with ssn fields smaller than 66 are in L. Similarly, if we were to look for 99, we would ignore L and focus on R because **all** nodes with ssns larger than 66 are in R.

Exercise 325. Design search-bst. The function consumes a number n and a BST. If the tree contains a node whose ssn field is n, the function produces the value of the name field in that node. Otherwise, the function produces NONE. The function organization must exploit the BST invariant so that the function performs as few comparisons as necessary.

See exercise 189 for searching in sorted lists. Compare!

Building a binary tree is easy; building a binary search tree is complicated. Given any two BTs, a number, and a name, we simply apply make-node to these values in the correct order, and voilà, we get a new BT. This same procedure fails for BSTs because the result would typically not be a BST. For example, if one BST contains nodes with ssn fields 3 and 5 in the correct order, and the other one contains ssn fields 2 and 6, simply combining the two trees with another social security number and a name does not produce a BST.

The remaining two exercises explain how to build a BST from a list of numbers and names. Specifically, the first exercise calls for a function that inserts a given ssn0 and name0 into a BST; that is, it produces a BST like the one it is given with one more node inserted containing ssn0, name0, and NONE subtrees. The second exercise then requests a function that can deal with a complete list of numbers and names.

Exercise 326. Design the function create-bst. It consumes a BST B, a number N, and a symbol S. It produces a BST that is just like B and that in place of one NONE subtree contains the node structure

```
(make-node N S NONE NONE)
```

Once the design is completed, use the function on tree A from figure 119.

Exercise 327. Design the function create-bst-from-list. It consumes a list of numbers and names and produces a BST by repeatedly applying create-bst. Here is the signature:

```
; [List-of [List Number Symbol]] -> BST
```

Use the complete function to create a BST from this sample input:

```
'((99 o)
  (77 l)
  (24 i)
  (10 h)
  (95 g)
  (15 d)
  (89 c)
  (29 b)
  (63 a))
```

The result is tree A in figure 119, if you follow the structural design recipe. If you use an existing abstraction, you may still get this tree but you may also get an “inverted” one. Why?

19.6 Simplifying Functions

Exercise 317 shows how to use local to organize a function that deals with an intertwined form of data. This organization also helps simplify functions once we know that the data definition is final. To demonstrate this point, we explain how to simplify the solution of exercise 319.

```
; S-expr Symbol Atom -> S-expr
; replaces all occurrences of old in sexp with new

(check-expect (substitute '(((world) bye) bye) 'bye '42)
              '(((world) 42) 42))

(define (substitute sexp old new)
  (local (; S-expr -> S-expr
          (define (for-sexp sexp)
            (cond
              [(atom? sexp) (for-atom sexp)]
              [else (for-sl sexp)]))
          ; SL -> S-expr
          (define (for-sl sl)
            (cond
```

```
              [(empty? sl) '()]
              [else (cons (for-sexp (first sl))
                          (for-sl (rest sl)))]))
          ; Atom -> S-expr
          (define (for-atom at)
            (cond
              [(number? at) at]
              [(string? at) at]
              [(symbol? at) (if (equal? at old) new at)])))
    (for-sexp sexp)))
```

Figure 120: A program to be simplified

Figure 120 displays a complete definition of the `substitute` function. The definition uses `local` and three auxiliary functions as suggested by the data definition. The figure includes a test case so that you can retest the function after each edit suggested below. Stop! Develop additional test cases; one is almost never enough.

Exercise 328. Copy and paste figure 120 into DrRacket; include your test suite. Validate the test suite. As you read along the remainder of this section, perform the edits and rerun the test suites to confirm the validity of our arguments.

```
(define (substitute sexp old new)
  (local (; S-expr -> S-expr
          (define (for-sexp sexp)
            (cond
              [(atom? sexp) (for-atom sexp)]
              [else (for-sl sexp)]))
          ; SL -> S-expr
          (define (for-sl sl)
            (map for-sexp sl))
          ; Atom -> S-expr
          (define (for-atom at)
            (cond
              [(number? at) at]
              [(string? at) at]
              [(symbol? at) (if (equal? at old) new at)])))
    (for-sexp sexp)))
```

Figure 121: Program simplification, step 1

Since we know that SL describes lists of S-expr, we can use `map` to simplify `for-sl`. See figure 121 for the result. While the original program says that `for-sexp` is applied to every item on `sl`, its revised definition expresses the same idea more succinctly with `map`.

For the second simplification step, we need to remind you that `equal?` compares two arbitrary values. With this in mind, the third `local` function becomes a one-liner. Figure 122 displays this second simplification.

```
(define (substitute sexp old new)
  (local (; S-expr -> S-expr
          (define (for-sexp sexp)
            (cond
              [(atom? sexp) (for-atom sexp)]
              [else (for-sl sexp)]))
          ; SL -> S-expr
          (define (for-sl sl) (map for-sexp sl))
          ; Atom -> S-expr
          (define (for-atom at)
            (if (equal? at old) new at)))
    (for-sexp sexp)))

(define (substitute.v3 sexp old new)
  (local (; S-expr -> S-expr
          (define (for-sexp sexp)
            (cond
              [(atom? sexp)
               (if (equal? sexp old) new sexp)]
              [else
               (map for-sexp sexp)])))
    (for-sexp sexp)))
```

Figure 122: Program simplification, steps 2 and 3

At this point the last two `local` definitions consist of a single line. Furthermore, neither definition is recursive. Hence we can *in-line* the functions in `for-sexp`. In-lining means replacing `(for-atom sexp)` with `(if (equal? sexp old) new sexp)`, that is, we replace the parameter `at` with the actual argument `sexp`. Similarly, for `(for-sl sexp)` we put in `(map for-sexp sexp)`; see the bottom half of figure 121. All we are left with now is a function whose definition introduces one `local` function, which is called on the same major argument. If we systematically supplied the other two arguments, we would immediately see that the locally defined function can be used in lieu of the outer one.

> While `sexp` is also a parameter, this substitution is really acceptable because it, too, stands in for an actual value.

Here is the result of translating this last thought into code:

```
(define (substitute sexp old new)
  (cond
    [(atom? sexp) (if (equal? sexp old) new sexp)]
    [else
     (map (lambda (s) (substitute s old new)) sexp)]))
```

Stop! Explain why we had to use `lambda` for this last simplification.

20 Iterative Refinement

When you develop real-world programs, you may confront complex forms of information and the problem of representing them with data. The best strategy to approach this task is to use *iterative refinement*, a well-known scientific process. A scientist's problem is to represent a part of the real world, using some form of mathematics. The result of the effort is called a model. The scientist then tests the model in many ways, in particular by predicting the outcome of experiments. If the discrepancies between the predictions and the measurements are too large, the model is refined with the goal of improving the predictions. This iterative process continues until the predictions are sufficiently accurate.

Consider a physicist who wishes to predict a rocket's flight path. While a "rocket as a point" representation is simple, it is also quite inaccurate, failing to account for air friction, for example. In response, the physicist may add the rocket's rough contour and introduce the necessary mathematics to represent friction. This second model is a *refinement* of the first model. In general, a scientist repeats—or as programmers say, *iterates*—this process until the model predicts the rocket's flight path with sufficient accuracy.

A programmer trained in a computer science department should proceed like this physicist. The key is to find an accurate data representation of the real-world information and functions that process them appropriately. Complicated situations call for a refinement process to get to a sufficient data representation combined with the proper functions. The process starts with the essential pieces of information and adds others as needed. Sometimes a programmer must refine a model **after** the program has been deployed because users request additional functionality.

So far we have used iterative refinement for you when it came to complex forms of data. This chapter illustrates iterative refinement as a principle of program development with an extended example, representing and processing (portions of) a computer's file system. We start with a brief discussion of the file system and then iteratively develop three data representations. Along the way, we propose some programming exercises so that you see how the design recipe also helps modify existing programs.

20.1 Data Analysis

Before you turn off DrRacket, you want to make sure that all your work is safely stashed away somewhere. Otherwise you have to reenter everything when you fire up DrRacket next. So you ask your computer to save programs and data in *files*. A file is roughly a string.

> A file is really a sequence of *bytes*, one after another. Try to define the class of files.

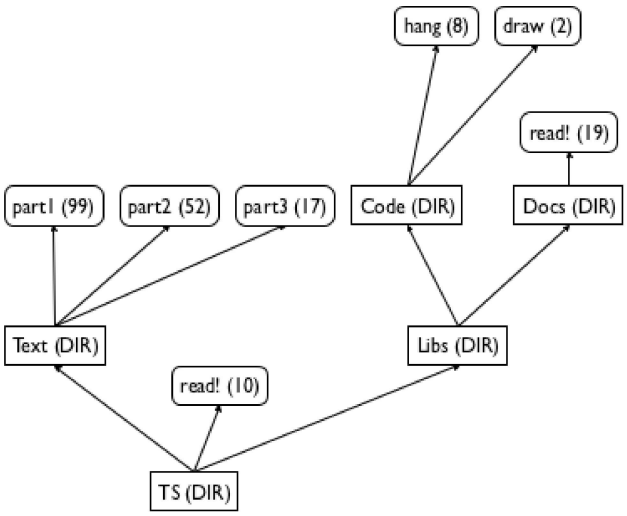


Figure 123: A sample directory tree

On most computer systems, files are organized in *directories* or *folders*. Roughly speaking, a directory contains some files and some more directories. The latter are called sub-directories and may contain yet more sub-directories and files. Because of the hierarchy, we speak of *directory trees*.

Figure 123 contains a graphical sketch of a small directory tree, and the picture explains why computer scientists call them trees. Contrary to convention in computer science, the figure shows the tree growing upward, with a root directory named `TS`. The root directory contains one file, called `read!`, and two sub-directories, called `Text` and `Libs`, respectively. The first sub-directory, `Text`, contains only three files; the latter, `Libs`, contains only two sub-directories, each of which contains at least one file. Finally, each box has one of two annotations: a directory is annotated with `DIR`, and a file is annotated with a number, its size.

Exercise 329. How many times does a file name `read!` occur in the directory tree `TS`? Can you describe the path from the root directory to the occurrences? What is the total size of all the files in the tree? What is the total size of the directory if each directory node has size `1`? How many levels of directories does it contain?

20.2 Refining Data Definitions

Exercise 329 lists some of the questions that users routinely ask about directories. To answer such questions, the computer's operating system provides programs that can answer them. If you want to design such programs, you need to develop a data representation for directory trees.

In this section, we use iterative refinement to develop three such data representations. For each stage, we need to decide which attributes to include and which to ignore. Consider the directory tree in figure 123 and imagine how it is created. When a user first creates a directory, it is empty. As time goes by, the user adds files and directories. In general, a user refers to files by names but mostly thinks of directories as containers.

Model 1 Our thought experiment suggests that our first model should focus on files as atomic entities with a name and directories as containers. Here is a data definition that deals with directories as lists and files as strings, that is, their names:

```
; A Dir.v1 (short for directory) is one of:
; – '()
; – (cons File.v1 Dir.v1)
; – (cons Dir.v1 Dir.v1)

; A File.v1 is a String.
```

The names have a *.v1* suffix to distinguish them from future refinements.

Exercise 330. Translate the directory tree in figure 123 into a data representation according to model 1.

Exercise 331. Design the function `how-many`, which determines how many files a given Dir.v1 contains. Remember to follow the design recipe; exercise 330 provides you with data examples.

Model 2 If you solved exercise 331, you know that this first data definition is still reasonably simple. But, it also obscures the nature of directories. With this first representation, we would not be able to list all the names of the sub-directories of some given directory. To model directories in a more faithful manner than containers, we must introduce a structure type that combines a name with a container:

```
(define-struct dir [name content])
```

This new structure type, in turn, suggests the following revision of the data definition:

```
; A Dir.v2 is a structure:
;   (make-dir String LOFD)

; An LOFD (short for list of files and directories) is one of:
; – '()
; – (cons File.v2 LOFD)
; – (cons Dir.v2 LOFD)

; A File.v2 is a String.
```

Note how the data definition for Dir.v2 refers to the definition for LOFD,s and the one for LOFDs refers back to that of Dir.v2. The two definitions are mutually recursive.

Exercise 332. Translate the directory tree in figure 123 into a data representation according to model 2.

Exercise 333. Design the function `how-many`, which determines how many files a given Dir.v2 contains. Exercise 332 provides you with data examples. Compare your result with that of exercise 331.

Exercise 334. Show how to equip a directory with two more attributes: size and readability. The former measures how much space the directory itself (as opposed to its content) consumes; the latter specifies whether anyone else besides the user may browse the content of the directory.

Model 3 Like directories, files have attributes. To introduce these, we proceed just as above. First, we define a structure for files:

```
(define-struct file [name size content])
```

Second, we provide a data definition:

```
; A File.v3 is a structure:
;   (make-file String N String)
```

As indicated by the field names, the string represents the name of the file, the natural number its size, and the string its content.

Finally, let's split the `content` field of directories into two pieces: a list of files and a list of sub-directories. This change requires a revision of the structure type definition:

```
(define-struct dir.v3 [name dirs files])
```

Here is the refined data definition:

```
; A Dir.v3 is a structure:
;   (make-dir.v3 String Dir* File*)

; A Dir* is one of:
; – '()
; – (cons Dir.v3 Dir*)

; A File* is one of:
; – '()
; – (cons File.v3 File*)
```

Following a convention in computer science, the use of `*` as the ending of a name suggests "many" and is a marker distinguishing the name from similar ones: File.v3 and Dir.v3.

Exercise 335. Translate the directory tree in figure 123 into a data representation according to model 3. Use `""` for the content of files.

Exercise 336. Design the function `how-many`, which determines how many files a given Dir.v3 contains. Exercise 335 provides you with data examples. Compare your result with that of exercise 333.

Given the complexity of the data definition, contemplate how anyone can design correct functions. Why are you confident that `how-many` produces correct results?

Exercise 337. Use List-of to simplify the data definition Dir.v3. Then use ISL+'s list-processing functions from figures 95 and 96 to simplify the function definition(s) for the solution of exercise 336.

Starting with a simple representation of the first model and refining it step-by-step, we have developed a reasonably accurate data representation for directory trees. Indeed, this third data representation captures the nature of a directory

tree much more faithfully than the first two. Based on this model, we can create a number of other functions that users expect from a computer's operating system.

20.3 Refining Functions

To make the following exercises somewhat realistic, DrRacket comes with the *dir.rkt* library from the first edition of this book. This teachpack introduces the two structure type definitions from model 3, though without the *.v3* suffix. Furthermore, the teachpack provides a function that creates representations of directory trees on your computer:

> Add `(require htdp/dir)` to the definitions area.

```
; String -> Dir.v3
; creates a representation of the a-path directory
(define (create-dir a-path) ...)
```

For example, if you open DrRacket and enter the following three lines into the definitions area:

```
(define O (create-dir "/Users/...")) ; on OS X
(define L (create-dir "/var/log/")) ; on Linux
(define W (create-dir "C:\\Users\\...")) ; on Windows
```

you get data representations of directories on your computer after you **save** and then run the program. Indeed, you could use `create-dir` to map the entire file system on your computer to an instance of Dir.v3.

Warnings (1) For large directory trees, DrRacket may need a lot of time to build a representation. Use `create-dir` on small directory trees first. (2) Do **not** define your own *dir* structure type. The teachpack already defines them, and you must not define a structure type twice.

Although `create-dir` delivers only a representation of a directory tree, it is sufficiently realistic to give you a sense of what it is like to design programs at that level. The following exercises illustrate this point. They use *Dir* to refer to the generic idea of a data representation for directory trees. Use the simplest data definition of *Dir* that allows you to complete the respective exercise. Feel free to use the data definition from exercise 337 and the functions from figures 95 and 96.

Exercise 338. Use `create-dir` to turn some of your directories into ISL+ data representations. Then use `how-many` from exercise 336 to count how many files they contain. Why are you confident that `how-many` produces correct results for these directories?

Exercise 339. Design `find?`. The function consumes a *Dir* and a file name and determines whether or not a file with this name occurs in the directory tree.

Exercise 340. Design the function `ls`, which lists the names of all files and directories in a given *Dir*.

Exercise 341. Design `du`, a function that consumes a *Dir* and computes the total size of all the files in the entire directory tree. Assume that storing a directory in a *Dir* structure costs `1` file storage unit. In the real world, a directory is basically a special file, and its size depends on how large its associated directory is.

The remaining exercises rely on the notion of a path, which for our purposes is a list of names:

```
; A Path is [List-of String].
; interpretation directions into a directory tree
```

Take a second look at figure 123. In that diagram, the path from `TS` to `part1` is `(list "TS" "Text" "part1")`. Similarly, the path from `TS` to `Code` is `(list "TS" "Libs" "Code")`.

Exercise 342. Design `find`. The function consumes a directory `d` and a file name `f`. If `(find? d f)` is `#true`, `find` produces a path to a file with name `f`; otherwise it produces `#false`.

Hint While it is tempting to first check whether the file name occurs in the directory tree, you have to do so for every single sub-directory. Hence it is better to combine the functionality of `find?` and `find`.

Challenge The `find` function discovers only one of the two files named `read!` in figure 123. Design `find-all`, which generalizes `find` and produces the list of all paths that lead to `f` in `d`. What should `find-all` produce when `(find? d f)` is `#false`? Is this part of the problem really a challenge compared to the basic problem?

Exercise 343. Design the function `ls-R`, which lists the paths to **all** files contained in a given *Dir*.

Exercise 344. Redesign `find-all` from exercise 342 using `ls-R` from exercise 343. This is design by composition, and if you solved the challenge part of exercise 342 your new function can find directories, too.

21 Refining Interpreters

DrRacket is a program. It is a complex one, dealing with many different kinds of data. Like most complex programs, DrRacket also consists of many functions: one that allows programmers to edit text, another one that acts like the interactions area, a third one that checks whether definitions and expressions are “grammatical,” and so on.

In this chapter, we show you how to design the function that implements the heart of the interactions area. Naturally, we use iterative refinement for this design project. As a matter of fact, the very idea of focusing on this aspect of DrRacket is another instance of refinement, namely, the obvious one of implementing only one piece of functionality.

Simply put, the interactions area performs the task of determining the values of expressions that you enter. After you click *RUN*, the interactions area knows about all the definitions. It is then ready to accept an expression that may refer to these definitions, to determine the value of this expression, and to repeat this cycle as often as you wish. For this reason, many people also refer to the interactions area as the *read-eval-print* loop, where *eval* is short for *evaluator*, a function that is also called *interpreter*.

Like this book, our refinement process starts with numeric BSL expressions. They are simple; they do not assume an understanding of definitions; and even your sister in fifth grade can determine their value. Once you understand this first step, you know the difference between a BSL expression and its representation. Next we move on to expressions with variables. The last step is to add definitions.

21.1 Interpreting Expressions

Our first task is to agree on a data representation for BSL programs. That is, we must figure out how to represent a BSL expression as a piece of BSL data. At first, this sounds strange and unusual, but it is not difficult. Suppose we just want to represent numbers, additions, and multiplications for a start. Clearly, numbers can stand for numbers. An addition expression, however, calls for compound data because it contains two expressions and because it is distinct from a multiplication expression, which also needs a data representation.

Following Adding Structure, a straightforward way to represent additions and multiplications is to define two structure types, each with two fields:

```
(define-struct add [left right])
(define-struct mul [left right])
```

The intention is that the `left` field contains one operand—the one to the “left” of the operator—and the `right` field contains the other operand. The following table shows three examples:

BSL expression	representation of BSL expression
`3`	`3`
`(+ 1 1)`	`(make-add 1 1)`
`(* 300001 100000)`	`(make-mul 300001 100000)`

The next question concerns an expression with sub-expressions:

```
(+ (* 3 3) (* 4 4))
```

The surprisingly simple answer is that fields may contain any value. In this particular case, `left` and `right` may contain representations of expressions, and you may nest this as deeply as you wish. See figure 124 for additional examples.

BSL expression	representation of BSL expression
`(+ (* 1 1) 10)`	`(make-add (make-mul 1 1) 10)`
`(+ (* 3 3)` `   (* 4 4))`	`(make-add (make-mul 3 3)` `           (make-mul 4 4))`

`(+ (* (+ 1 2) 3)`
`   (* (* (+ 1 1)`
`         2)`
`      4))`
	`(make-add (make-mul (make-add 1 2) 3)`
	`           (make-mul (make-mul`
	`                       (make-add 1 1)`
	`                       2)`
	`                     4))`

Figure 124: Representing BSL expressions in BSL

Exercise 345. Formulate a data definition for the representation of BSL expressions based on the structure type definitions of `add` and `mul`. Let’s use *BSL-expr* in analogy for S-expr for the new class of data.

Translate the following expressions into data:

1. `(+ 10 -10)`

2. `(+ (* 20 3) 33)`

3. `(+ (* 3.14 (* 2 3)) (* 3.14 (* -1 -9)))`

Interpret the following data as expressions:

1. `(make-add -1 2)`

2. `(make-add (make-mul -2 -3) 33)`

3. `(make-mul (make-add 1 (make-mul 2 3))
   3.14)`

> Here "interpret" means "translate from data into information." In contrast, "interpreter" in the title of this chapter refers to a program that consumes the representation of a program and produces its value. While the two ideas are related, they are not the same.

Now that you have a data representation for BSL programs, it is time to design an evaluator. This function consumes a representation of a BSL expression and produces its value. Again, this function is unlike any you have ever designed so it pays off to experiment with some examples. To this end, either you can use the rules of arithmetic to figure out what the value of an expression is or you can "play" in the interactions area of DrRacket. Take a look at the following table for our examples:

BSL expression	its representation	its value
`3`	`3`	`3`
`(+ 1 1)`	`(make-add 1 1)`	`2`
`(* 3 10)`	`(make-mul 3 10)`	`30`
`(+ (* 1 1) 10)`	`(make-add (make-mul 1 1) 10)`	`11`

Exercise 346. Formulate a data definition for the class of values to which a representation of a BSL expression can evaluate.

Exercise 347. Design `eval-expression`. The function consumes a representation of a BSL expression and computes its value.

Exercise 348. Develop a data representation for Boolean BSL expressions constructed from `#true`, `#false`, `and`, `or`, and `not`. Then design `eval-bool-expression`, which consumes (representations of) Boolean BSL expressions and computes their values. What kind of values do these Boolean expressions yield?

Convenience and parsing S-expressions offer a convenient way to represent BSL expressions in our programming language:

```
> (+ 1 1)
2
> '(+ 1 1)
(list '+ 1 1)
> (+ (* 3 3) (* 4 4))
25
> '(+ (* 3 3) (* 4 4))
(list '+ (list '* 3 3) (list '* 4 4))
```

By simply putting a quote in front of an expression, we get ISL+ data.

Interpreting an S-expression representation is clumsy, mostly because not all S-expressions represent BSL-exprs. For example, `#true`, `"hello"`, and `'(+ x 1)` are not representatives of BSL expressions. As a result, S-expressions are quite inconvenient for the designers of interpreters.

Programmers invented *parsers* to bridge the gap between convenience of use and implementation. A parser simultaneously checks whether some piece of data conforms to a data definition and, if it does, builds a matching element from the chosen class of data. The latter is called a *parse tree*. If the given data does not conform, a parser signals an error, much like the checked functions from Input Errors.

Figure 125 presents a BSL parser for S-expressions. Specifically, `parse` consumes an S-expr and produces a `BSL-expr`—if and only if the given S-expression is the result of quoting a BSL expression that has a `BSL-expr` representative.

Exercise 349. Create tests for `parse` until DrRacket tells you that every element in the definitions area is covered during the test run.

Exercise 350. What is unusual about the definition of this program with respect to the design recipe?

Note One unusual aspect is that `parse` uses `length` on the list argument. Real parsers avoid `length` because it slows the functions down.

Exercise 351. Design `interpreter-expr`. The function accepts S-expressions. If `parse` recognizes them as BSL-expr, it produces their value. Otherwise, it signals the same error as `parse`.

```
; S-expr -> BSL-expr
(define (parse s)
```

```
  (cond
    [(atom? s) (parse-atom s)]
    [else (parse-sl s)]))

; SL -> BSL-expr
(define (parse-sl s)
  (local ((define L (length s)))
    (cond
      [(< L 3) (error WRONG)]
      [(and (= L 3) (symbol? (first s)))
       (cond
         [(symbol=? (first s) '+)
          (make-add (parse (second s)) (parse (third s)))]
         [(symbol=? (first s) '*)
          (make-mul (parse (second s)) (parse (third s)))]
         [else (error WRONG)])]
      [else (error WRONG)])))

; Atom -> BSL-expr
(define (parse-atom s)
  (cond
    [(number? s) s]
    [(string? s) (error WRONG)]
    [(symbol? s) (error WRONG)]))
```

Figure 125: From S-expr to BSL-expr

21.2 Interpreting Variables

Since the first section ignores constant definitions, an expression does not have a value if it contains a variable. Indeed, unless we know what x stands for, it makes no sense to evaluate `(+ 3 x)`. Hence, one first refinement of the evaluator is to add variables to the expressions that we wish to evaluate. The assumption is that the definitions area contains a definition such as

```
(define x 5)
```

and that programmers evaluate expressions containing x in the interactions area:

```
> x
5
> (+ x 3)
8
> (* 1/2 (* x 3))
7.5
```

Indeed, you could imagine a second definition, say `(define y 3)`, and interactions that involve two variables:

```
> (+ (* x x)
     (* y y))
34
```

The preceding section implicitly proposes symbols as representations for variables. After all, if you were to choose quoted S-expressions to represent expressions with variables, symbols would appear naturally:

```
> 'x
'x
> '(* 1/2 (* x 3))
(list '* 0.5 (list '* 'x 3))
```

One obvious alternative is a string, so that `"x"` would represent x, but this book is not about designing interpreters, so we stick with symbols. From this decision, it follows how to modify the data definition from exercise 345:

```
; A BSL-var-expr is one of:
; – Number
; – Symbol
; – (make-add BSL-var-expr BSL-var-expr)
; – (make-mul BSL-var-expr BSL-var-expr)
```

We simply add one clause to the data definition.

As for data examples, the following table shows some BSL expressions with variables and their BSL-var-expr representation:

BSL expression	representation of BSL expression
`x`	`'x`
`(+ x 3)`	`(make-add 'x 3)`
`(* 1/2 (* x 3))`	`(make-mul 1/2 (make-mul 'x 3))`
`(+ (* x x)` `   (* y y))`	`(make-add (make-mul 'x 'x)` `           (make-mul 'y 'y))`

They are all taken from the interactions above, meaning you know the results when `x` is `5` and `y` `3`.

One way to determine the value of variable expressions is to replace all variables with the values that they represent. This is the way you know from mathematics classes in school, and it is a perfectly fine way.

Exercise 352. Design `subst`. The function consumes a BSL-var-expr `ex`, a Symbol `x`, and a Number `v`. It produces a BSL-var-expr like `ex` with all occurrences of `x` replaced by `v`.

Exercise 353. Design the `numeric?` function. It determines whether a BSL-var-expr is also a BSL-expr. Here we assume that your solution to exercise 345 is the definition for BSL-var-expr without Symbols.

Exercise 354. Design `eval-variable`. The checked function consumes a `BSL-var-expr` and determines its value if `numeric?` yields true for the input. Otherwise it signals an error.

In general, a program defines many constants in the definitions area, and expressions contain more than one variable. To evaluate such expressions, we need a representation of the definitions area when it contains a series of constant definitions. For this exercise we use association lists:

```
; An AL (short for association list) is [List-of Association].
; An Association is a list of two items:
;   (cons Symbol (cons Number '())).
```

Make up elements of AL.

Design `eval-variable*`. The function consumes a BSL-var-expr `ex` and an association list `da`. Starting from `ex`, it iteratively applies `subst` to all associations in `da`. If `numeric?` holds for the result, it determines its value; otherwise it signals the same error as `eval-variable`. **Hint** Think of the given BSL-var-expr as an atomic value and traverse the given association list instead. We provide this hint because the creation of this function requires a little design knowledge from Simultaneous Processing.

An environment model Exercise 354 relies on the mathematical understanding of constant definitions. If a name is defined to stand for some value, all occurrences of the name can be replaced with the value. Substitution performs this replacement once and for all before the evaluation process even starts.

An alternative approach, dubbed the *environment model*, is to look up the value of a variable when needed. The evaluator starts processing the expression immediately but also carries along the representation of the definitions area. Every time the evaluator encounters a variable, it looks in the definitions area for its value and uses it.

Exercise 355. Design `eval-var-lookup`. This function has the same signature as `eval-variable*`:

```
; BSL-var-expr AL -> Number
(define (eval-var-lookup e da) ...)
```

Instead of using substitution, the function traverses the expression in the manner that the design recipe for BSL-var-expr suggests. As it descends the expression, it “carries along” `da`. When it encounters a symbol `x`, it uses `assq` to look up the value of `x` in the association list. If there is no value, `eval-var-lookup` signals an error.

21.3 Interpreting Functions

At this point, you understand how to evaluate BSL programs that consist of constant definitions and variable expressions. Naturally you want to add function definitions so that you know—at least in principle—how to deal with all of BSL.

The goal of this section is to refine the evaluator of Interpreting Variables so that it can cope with functions. Since function definitions show up in the definitions area, another way to describe the refined evaluator is to say that it simulates DrRacket when the definitions area contains a number of function definitions and a programmer enters an expression in the interactions area that contains uses of these functions.

For simplicity, let’s assume that all functions in the definitions area consume one argument and that there is only one such definition. The necessary domain knowledge dates back to school where you learned that $f(x) = e$ represents the definition of function f, that $f(a)$ represents the application of f to a, and that to evaluate the latter, you substitute a for x in e. As it turns out, the evaluation of function applications in a language such as BSL works mostly like that, too.

Before tackling the following exercises, you may wish to refresh your knowledge of the terminology concerning functions as presented in intermezzo 1. Most of the time, algebra courses gloss over this aspect of mathematics, but a precise use and understanding of terminology is needed when you wish to solve these problems.

Exercise 356. Extend the data representation of Interpreting Variables to include the application of a programmer-defined function. Recall that a function application consists of two pieces: a name and an expression. The former is the name of the function that is applied; the latter is the argument.

Represent these expressions: `(k (+ 1 1))`, `(* 5 (k (+ 1 1)))`, `(* (i 5) (k (+ 1 1)))`. We refer to this newly defined class of data as *BSL-fun-expr*.

Exercise 357. Design `eval-definition1`. The function consumes four arguments:

1. a BSL-fun-expr `ex`;

2. a symbol `f`, which represents a function name;

3. a symbol `x`, which represents the functions's parameter; and

4. a BSL-fun-expr `b`, which represents the function's body.

It determines the value of `ex`. When `eval-definition1` encounters an application of `f` to some argument, it

1. evaluates the argument,

2. substitutes the value of the argument for `x` in `b`; and

3. finally evaluates the resulting expression with `eval-definition1`.

Here is how to express the steps as code, assuming `arg` is the argument of the function application:

```
(local ((define value (eval-definition1 arg f x b))
        (define plugd (subst b x arg-value)))
  (eval-definition1 plugd f x b))
```

Notice that this line uses a form of recursion that has not been covered. The proper design of such functions is discussed in Generative Recursion.

If `eval-definition1` encounters a variable, it signals the same error as `eval-variable` from exercise 354. It also signals an error for function applications that refer to a function name other than `f`.

Warning The use of this uncovered form of recursion introduces a new element into your computations: non-termination. That is, a program may run forever instead of delivering a result or signaling an error. If you followed the design recipes of the first four parts, you cannot write down such programs. For fun, construct an input for `eval-definition1` that causes it to run forever. Use *STOP* to terminate the program.

For an evaluator that mimics the interactions area, we need a representation of the definitions area. We assume that it is a list of definitions.

Exercise 358. Provide a structure type and a data definition for function definitions. Recall that such a definition has three essential attributes:

1. the function's name, which is represented with a symbol;

2. the function's parameter, which is also a name; and

3. the function's body, which is a variable expression.

We use *BSL-fun-def* to refer to this class of data.

Use your data definition to represent these BSL function definitions:

1. `(define (f x) (+ 3 x))`

2. `(define (g y) (f (* 2 y)))`

3. `(define (h v) (+ (f v) (g v)))`

Next, define the class *BSL-fun-def** to represent a definitions area that consists of a number of one-argument function definitions. Translate the definitions area that defines `f`, `g`, and `h` into your data representation and name it `da-fgh`.

Finally, work on the following wish:

```
; BSL-fun-def* Symbol -> BSL-fun-def
; retrieves the definition of f in da
; signals an error if there is none
```

```
(check-expect (lookup-def da-fgh 'g) g)
(define (lookup-def da f) ...)
```

Looking up a definition is needed for the evaluation of applications.

Exercise 359. Design `eval-function*`. The function consumes `ex`, a BSL-fun-expr, and `da`, a BSL-fun-def* representation of a definitions area. It produces the result that DrRacket shows if you evaluate `ex` in the interactions area, assuming the definitions area contains `da`.

The function works like `eval-definition1` from exercise 357. For an application of some function `f`, it

1. evaluates the argument;

2. looks up the definition of `f` in the BSL-fun-def representation of `da`, which comes with a parameter and a body;

3. substitutes the value of the argument for the function parameter in the function's body; and

4. evaluates the new expression via recursion.

Like DrRacket, `eval-function*` signals an error when it encounters a variable or function name without definition in the definitions area.

21.4 Interpreting Everything

Take a look at the following BSL program:

```
(define close-to-pi 3.14)

(define (area-of-circle r)
  (* close-to-pi (* r r)))

(define (volume-of-10-cylinder r)
  (* 10 (area-of-circle r)))
```

Think of these definitions as the definitions area in DrRacket. After you click *RUN*, you can evaluate expressions involving `close-to-pi`, `area-of-circle`, and `volume-of-10-cylinder` in the interactions area:

```
> (area-of-circle 1)
#i3.14
> (volume-of-10-cylinder 1)
#i31.400000000000002
> (* 3 close-to-pi)
#i9.42
```

The goal of this section is to refine your evaluator again so that it can mimic this much of DrRacket.

Exercise 360. Formulate a data definition for the representation of DrRacket's definitions area. Concretely, the data representation should work for a sequence that freely mixes constant definitions and one-argument function definitions. Make sure you can represent the definitions area consisting of three definitions at the beginning of this section. We name this class of data *BSL-da-all*.

Design the function `lookup-con-def`. It consumes a BSL-da-all `da` and a symbol `x`. It produces the representation of a constant definition whose name is `x`, if such a piece of data exists in `da`; otherwise the function signals an error saying that no such constant definition can be found.

Design the function `lookup-fun-def`. It consumes a BSL-da-all `da` and a symbol `f`. It produces the representation of a function definition whose name is `f`, if such a piece of data exists in `da`; otherwise the function signals an error saying that no such function definition can be found.

Exercise 361. Design `eval-all`. Like `eval-function*` from exercise 359, this function consumes the representation of an expression and a definitions area. It produces the same value that DrRacket shows if the expression is entered at the prompt in the interactions area and the definitions area contains the appropriate definitions. **Hint** Your `eval-all` function should process variables in the given expression like `eval-var-lookup` in exercise 355.

Exercise 362. It is cumbersome to enter the structure-based data representation of BSL expressions and a definitions area. As the end of Interpreting Expressions demonstrates, it is much easier to quote expressions and (lists of) definitions.

Design a function `interpreter`. It consumes an `S-expr` and an `Sl`. The former is supposed to represent an expression and the latter a list of definitions. The function parses both with the appropriate parsing functions and then uses `eval-all` from exercise 361 to evaluate the expression. **Hint** You must adapt the ideas of exercise 350 to create a parser for definitions and lists of definitions.

You should know that `eval-all-sexpr` makes it straightforward to check whether it really mimics DrRacket's evaluator.

At this point, you know a lot about interpreting BSL. Here are some of the missing pieces: Booleans with `cond` or `if`; Strings and such operations `string-length` or `string-append`; and lists with `'()`, `empty?`, `cons`, `cons?`, `first`, `rest`; and so on. Once your evaluator can cope with all these, it is basically complete because your evaluators already know how to interpret recursive functions. Now when we say "trust us, you know how to design these refinements," we mean it.

22 Project: The Commerce of XML

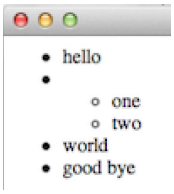
XML is a widely used data language. One use concerns message exchanges between programs running on different computers. For example, when you point your web browser at a web site, you are connecting a program on your computer to a program on another computer, and the latter sends XML data to the former. Once the browser receives the XML data, it renders it as an image on your computer's monitor.

The following comparison illustrates this idea with a concrete example:

XML data

```
<ul>
 <li> hello </li>
 <li> <ul>
     <li> one </li>
     <li> two </li>
    </ul>
 </li>
 <li> world </li>
 <li> good bye </li>
</ul>
```

rendered in a browser



On the left, you see a piece of XML data that a web site may send to your web browser. On the right, you see how one popular browser renders this snippet graphically.

This chapter explains the basics of processing XML as another design exercise concerning intertwined data definitions and iterative refinement. The next section starts with an informal comparison of S-expressions and XML data and uses it to formulate a full-fledged data definition. The remaining sections explain with examples how to process an S-expression of XML data.

> If you think XML is too old-fashioned for 2018, feel free to redo the exercise for JSON or some other modern data exchange format. The design principles remain the same.

22.1 XML as S-expressions

The most basic piece of XML data looks like this:

```
<machine> </machine>
```

It is called an *element* and "machine" is the name of the element. The two parts of the elements are like parentheses that delimit the *content* of an element. When there is no content between the two parts—other than white space—XML allows a short-hand:

```
<machine />
```

But, as far as we are concerned here, this short-hand is equivalent to the explicitly bracketed version.

From an S-expression perspective, an XML element is a **named** pair of parentheses that surround some content. And indeed, representing the above with an S-expression is quite natural:

```
'(machine)
```

> Racket's `xml` library represents XML with structures as well as S-expressions.

This piece of data has the opening and closing parentheses, and it comes with space to embed content.

Here is a piece of XML data with content:

```
<machine><action /></machine>
```

Remember that the `<action />` part is a short-hand, meaning we are really looking at this piece of data:

```
<machine><action></action></machine>
```

In general, the content of an XML element is a series of XML elements:

```
<machine><action /><action /><action /></machine>
```

Stop! Expand the short-hand for `<action />` before you continue.

The S-expression representation continues to look simple. Here is the first one:

```
'(machine (action))
```

And this is the representation for the second one:

```
'(machine (action) (action) (action))
```

When you look at the piece of XML data with a sequence of three `<action />` elements as its content, you realize that you may wish to distinguish such elements from each other. To this end, XML elements come with *attributes*. For example,

```
<machine initial="red"></machine>
```

is the "machine" element equipped with one attribute whose *name* is "initial" and whose value is "red" between string quotes. Here is a complex XML element with nested elements that have attributes, too:

```
<machine initial="red">
  <action state="red"    next="green" />
  <action state="green"  next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

We use blanks, indentation, and line breaks to make the element readable, but this white space has no meaning for our XML data here.

Naturally, S-expressions for these "machine" elements look much like their XML cousins:

```
'(machine ((initial "red")))
```

> XML is 40 years younger than S-expressions.

To add attributes to an element, we use a list of lists where each of the latter contains two items: a symbol and a string. The symbol represents the name of the attribute and the string its value. This idea naturally applies to complex forms of XML data, too:

```
'(machine ((initial "red"))
  (action ((state "red") (next "green")))
  (action ((state "green") (next "yellow")))
  (action ((state "yellow") (next "red"))))
```

For now note how the attributes are marked by two opening parentheses and the remaining list of (representations of) XML elements has one opening parenthesis.

You may recall the idea from Intermezzo 2: Quote, Unquote, which uses S-expressions to represent XHTML, a special dialect of XML. In particular, the intermezzo shows how easily a programmer can write down nontrivial XML data and even templates of XML representations using `backquote` and `unquote`. Of course, Interpreting Expressions points out that you need a parser to determine whether any given S-expression is a representation of XML data, and a parser is a complex and unusual kind of function.

Nevertheless, we choose to go with a representation of XML based on S-expressions to demonstrate the usefulness of this old, poetic idea in practical terms. We proceed gradually to work out a data definition, putting iterative refinement to work. Here is a first attempt:

```
; An Xexpr.v0 (short for X-expression) is a one-item list:
;   (cons Symbol '())
```

This is the "named parentheses" idea from the beginning of this section. Equipping this element representation with content is easy:

```
; An Xexpr.v1 is a list:
;   (cons Symbol [List-of Xexpr.v1])
```

The symbolic name becomes the first item on a list that otherwise consists of XML element representatives.

The last refinement step is to add attributes. Since the attributes in an XML element are optional, the revised data definition has two clauses:

```
; An Xexpr.v2 is a list:
; - (cons Symbol Body)
; - (cons Symbol (cons [List-of Attribute] Body))
; where Body is short for [List-of Xexpr.v2]
; An Attribute is a list of two items:
;   (cons Symbol (cons String '()))
```

Exercise 363. All elements of Xexpr.v2 start with a Symbol, but some are followed by a list of attributes and some by just a list of Xexpr.v2s. Reformulate the definition of Xexpr.v2 to isolate the common beginning and highlight the different kinds of endings.

Eliminate the use of List-of from Xexpr.v2.

Exercise 364. Represent this XML data as elements of Xexpr.v2:

1. `<transition from="seen-e" to="seen-f" />`

2. `<ul><li><word /><word /></li><li><word /></li></ul>`

Which one could be represented in Xexpr.v0 or Xexpr.v1?

Exercise 365. Interpret the following elements of Xexpr.v2 as XML data:

1. `'(server ((name "example.org")))`

2. `'(carcas (board (grass)) (player ((name "sam"))))`

3. `'(start)`

Which ones are elements of Xexpr.v0 or Xexpr.v1?

Roughly speaking, X-expressions simulate structures via lists. The simulation is convenient for programmers; it asks for the least amount of keyboard typing. For example, if an X-expression does not come with an attribute list, it is simply omitted. This choice of data representation represents a trade-off between authoring such expressions manually and processing them automatically. The best way to deal with the latter problem is to provide functions that make X-expressions look like structures, especially functions that access the quasi-fields:

- `xexpr-name`, which extracts the tag of the element representation;
- `xexpr-attr`, which extracts the list of attributes; and
- `xexpr-content`, which extracts the list of content elements.

Once we have these functions, we can use lists to represent XML yet that act as if they were instances of a structure type.

These functions parse S-expressions, and parsers are tricky to design. So let's design them carefully, starting with some data examples:

```
(define a0 '((initial "X")))

(define e0 '(machine))
(define e1 `(machine ,a0))
(define e2 '(machine (action)))
(define e3 '(machine () (action)))
(define e4 `(machine ,a0 (action) (action)))
```

The first definition introduces a list of attributes, which is reused twice in the construction of X-expressions. The definition of `e0` reminds us that an X-expression may not come with either attributes or content. You should be able to explain why `e2` and `e3` are basically equivalent.

Next we formulate a signature, a purpose statement, and a header:

```
; Xexpr.v2 -> [List-of Attribute]
; retrieves the list of attributes of xe
(define (xexpr-attr xe) '())
```

Here we focus on `xexpr-attr`; we leave the other two as exercises.

Making up functional examples requires a decision concerning the extraction of attributes from X-expressions without any. While our chosen representation completely omits missing attributes, we must supply `'()` for the structure-based representation of XML. The function therefore produces `'()` for such X-expressions:

```
(check-expect (xexpr-attr e0) '())
(check-expect (xexpr-attr e1) '((initial "X")))
(check-expect (xexpr-attr e2) '())
(check-expect (xexpr-attr e3) '())
(check-expect (xexpr-attr e4) '((initial "X")))
```

It is time to develop the template. Since the data definition for Xexpr.v2 is complex, we proceed slowly, step-by-step. First, while the data definition distinguishes two kinds of X-expressions, both clauses describe data constructed by `cons`ing a symbol onto a list. Second, what differentiates the two clauses is the rest of the list and especially the optional presence of a list of attributes. Let's translate these two insights into a template:

```
(define (xexpr-attr xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) ...]
```

```
       [else ...])))
```

The local definition chops off the name of the X-expression and leaves the remainder of the list, which may or may not start with a list of attributes. The key is that it is just a list, and the two `cond` clauses indicate so. Third, this list is **not** defined via a self-reference but as the optional `cons` of some attributes onto a possibly empty list of X-expressions. In other words, we still need to distinguish the two usual cases and extract the usual pieces:

```
(define (xexpr-attr xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) ...]
      [else (... (first optional-loa+content)
                 ... (rest optional-loa+content) ...)])))
```

At this point, we can already see that recursion is not needed for the task at hand. So, we switch to the fifth step of the design recipe. Clearly, there are no attributes if the given X-expression comes with nothing but a name. In the second clause, the question is whether the first item on the list is a list of attributes or just an Xexpr.v2. Because this sounds complicated, we make a wish:

```
; [List-of Attribute] or Xexpr.v2 -> ???
; determines whether x is an element of [List-of Attribute]
; #false otherwise
(define (list-of-attributes? x)
  #false)
```

With this function, it is straightforward to finish `xexpr-attr`; see figure 126. If the first item is a list of attributes, the function produces it; otherwise there are no attributes.

```
(define (xexpr-attr xe)
  (local ((define optional-loa+content (rest xe)))
    (cond
      [(empty? optional-loa+content) '()]
      [else
       (local ((define loa-or-x
                 (first optional-loa+content)))
         (if (list-of-attributes? loa-or-x)
             loa-or-x
             '()))])))
```

Figure 126: The complete definition of `xexpr-attr`

For the design of `list-of-attributes?`, we proceed in the same manner and get this definition:

```
; [List-of Attribute] or Xexpr.v2 -> Boolean
; is x a list of attributes
(define (list-of-attributes? x)
  (cond
    [(empty? x) #true]
    [else
     (local ((define possible-attribute (first x)))
       (cons? possible-attribute))]))
```

We skip the details of the design process because they are unremarkable. What is **remarkable** is the signature of this function. Instead of specifying a single data definition as possible inputs, the signature combines two data definitions separated by the English word “or.” In ISL+ such an informal signature with a definite meaning is acceptable on occasion.

Exercise 366. Design `xexpr-name` and `xexpr-content`.

Exercise 367. The design recipe calls for a self-reference in the template for `xexpr-attr`. Add this self-reference to the template and then explain why the finished parsing function does not contain it.

Exercise 368. Formulate a data definition that replaces the informal “or” signature for the definition of the `list-of-attributes?` function.

Exercise 369. Design `find-attr`. The function consumes a list of attributes and a symbol. If the attributes list associates the symbol with a string, the function retrieves this string; otherwise it returns `#false`.—Consider using `assq` to define the function.

For the remainder of this chapter, *Xexpr* refers to Xexpr.v2. Also, we assume `xexpr-name`, `xexpr-attr`, and `xexpr-content` are defined. Finally, we use `find-attr` from exercise 369 to retrieve attribute values.

22.2 Rendering XML Enumerations

XML is actually a **family** of languages. People define dialects for specific channels of communication. For example, XHTML is the language for sending web content in XML format. In this section, we illustrate how to design a rendering function for a small snippet of XHTML, specifically the enumerations from the beginning of this chapter.

The `ul` tag surrounds a so-called unordered HTML list. Each item of this list is tagged with `li`, which tends to contain words but also other elements, even enumerations. With "unordered" HTML means is that each item is to be rendered with a leading bullet instead of a number.

Since Xexpr does not come with plain strings, it is not immediately obvious how to represent XHTML enumerations in a subset. One option is to refine the data representation one more time, so that an Xexpr could be a String. Another option is to introduce a representation for text:

```
; An XWord is '(word ((text String))).
```

Here, we use this second option; Racket, the language from which the teaching languages are derived, offers libraries that include String in Xexpr.

Exercise 370. Make up three examples for XWords. Design `word?`, which checks whether some ISL+ value is in XWord, and `word-text`, which extracts the value of the only attribute of an instance of XWord.

Exercise 371. Refine the definition of Xexpr so that you can represent XML elements, including items in enumerations, that are plain strings.

Given the representation of words, representing an XHTML-style enumeration of words is straightforward:

```
; An XEnum.v1 is one of:
; – (cons 'ul [List-of XItem.v1])
; – (cons 'ul (cons Attributes [List-of XItem.v1]))
; An XItem.v1 is one of:
; – (cons 'li (cons XWord '()))
; – (cons 'li (cons Attributes (cons XWord '())))
```

For completeness, the data definition includes attribute lists, even though they do not affect rendering.

Stop! Argue that every element of XEnum.v1 is also in XExpr.

Here is a sample element of XEnum.v1:

```
(define e0
  '(ul
    (li (word ((text "one"))))
    (li (word ((text "two"))))))
```

It corresponds to the inner enumeration of the example from the beginning of the chapter. Rendering it with help from the *2htdp/image* library should yield an image like this:

The radius of the bullet and the distance between the bullet and the text are matters of aesthetics; here the idea matters.

To create this kind of image, you might use this ISL+ program:

> We developed these expressions in the interactions area. What would *you* do?

```
(define e0-rendered
  (above/align
   'left
   (beside/align 'center BT (text "one" 12 'black))
   (beside/align 'center BT (text "two" 12 'black))))
```

assuming `BT` is a rendering of a bullet.

Now let's design the function carefully. Since the data representation requires two data definitions, the design recipe tells you that you must design two functions in parallel. A second look reveals, however, that in this particular case the second data definition is disconnected from the first one, meaning we can deal with it separately.

Furthermore, the definition for XItem.v1 consists of two clauses, meaning the function itself should consist of a `cond` with two clauses. The point of viewing XItem.v1 as a sub-language of Xexpr, however, is to think of these two clauses in terms

of Xexpr selector functions, in particular, `xexpr-content`. With this function we can extract the textual part of an item, regardless of whether it comes with attributes or not:

```
; XItem.v1 -> Image
; renders an item as a "word" prefixed by a bullet
(define (render-item1 i)
  (... (xexpr-content i) ...))
```

In general, `xexpr-content` extracts a list of Xexpr; in this specific case, the list contains exactly one XWord, and this word contains one text:

```
(define (render-item1 i)
  (local ((define content (xexpr-content i))
          (define element (first content))
          (define a-word (word-text element)))
    (... a-word ...)))
```

From here, it is straightforward:

```
(define (render-item1 i)
  (local ((define content (xexpr-content i))
          (define element (first content))
          (define a-word (word-text element))
          (define item (text a-word 12 'black)))
    (beside/align 'center BT item)))
```

After extracting the text to be rendered in the item, it is simply a question of rendering it as text and equipping it with a leading bullet; see the examples above for how you might discover this last step.

Exercise 372. Before you read on, equip the definition of `render-item1` with tests. Make sure to formulate these tests in such a way that they don't depend on the `BT` constant. Then explain **how** the function works; keep in mind that the purpose statement explains only **what** it does.

Now we can focus on the design of a function that renders an enumeration. Using the example from above, the first two design steps are easy:

```
; XEnum.v1 -> Image
; renders a simple enumeration as an image
(check-expect (render-enum1 e0) e0-rendered)
(define (render-enum1 xe) empty-image)
```

The key step is the development of a template. According to the data definition, an element of XEnum.v1 contains one interesting piece of data, namely, the (representation of the) XML elements. The first item is always `'ul`, so there is no need to extract it, and the second, optional item is a list of attributes, which we ignore. With this in mind, the first template draft looks just like the one for `render-item1`:

```
(define (render-enum1 xe)
  (... (xexpr-content xe) ...)) ; [List-of XItem.v1]
```

While the data-oriented design recipe tells you that you should design a separate function whenever you encounter a complex form of data, the abstraction-based design recipe from Abstraction tells you to reuse an existing abstraction, say, a list-processing function from figures 95 and 96, when possible. Given that `render-enum1` is supposed to process a list and create a single image from it, the only two list-processing abstractions whose signatures fit the bill are `foldr` and `foldl`. If you also study their purpose statements, you see a pattern that is like the `e0-rendered` example above, especially for `foldr`. Let's try to use it, following the reuse design recipe:

```
(define (render-enum1 xe)
   (local ((define content (xexpr-content xe))
           ; XItem.v1 Image -> Image
           (define (deal-with-one item so-far)
             ...))
     (foldr deal-with-one empty-image content)))
```

From the type matching, you also know that:

1. the first argument to `foldr` must be a two-argument function;

2. the second argument must be an image; and

3. the last argument is the list representing XML content.

Naturally `empty-image` is the correct starting point.

This design-by-reuse focuses our attention on the function to be "folded" over the list. It turns one item and the image that `foldr` has created so far into another image. The signature for `deal-with-one` articulates this insight. Since the first argument is an XItem.v1, `render-item1` is the function that renders it. This yields two images that must be combined: the image of the first item and the image of the rest of the items. To stack them, we use `above`:

```
(define (render-enum1 xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v1 Image -> Image
          (define (deal-with-one item so-far)
            (above/align 'left
                         (render-item1 item)
                         so-far)))
    (foldr deal-with-one empty-image content)))
```

```
; An XItem.v2 is one of:
; – (cons 'li (cons XWord '()))
; – (cons 'li (cons [List-of Attribute] (list XWord)))
; – (cons 'li (cons XEnum.v2 '()))
; – (cons 'li (cons [List-of Attribute] (list XEnum.v2)))
;
; An XEnum.v2 is one of:
; – (cons 'ul [List-of XItem.v2])
; – (cons 'ul (cons [List-of Attribute] [List-of XItem.v2]))
```

Figure 127: A realistic data representation of XML enumerations

Flat enumerations are common, but they are also a simple approximation of the full-fledged case. In the real world, web browsers must cope with arbitrarily nested enumerations that arrive over the web. In XML and its web browser dialect XHTML, nesting is straightforward. Any element may show up as the content of any other element. To represent this relationship in our limited XHTML representation, we say that an item is either a word or another enumeration. Figure 127 displays the second revision of the data definition. It includes a revision of the data definition for enumerations so that the first definition refers to the correct form of item.

> Are you wondering whether arbitrary nesting is the correct way to think about this problem? If so, develop a data definition that allows only three levels of nesting and then use it.

```
(define SIZE 12) ; font size
(define COLOR "black") ; font color
(define BT ; a graphical constant
  (beside (circle 1 'solid 'black) (text " " SIZE COLOR)))

; Image -> Image
; marks item with bullet
(define (bulletize item)
  (beside/align 'center BT item))

; XEnum.v2 -> Image
; renders an XEnum.v2 as an image
(define (render-enum xe)
  (local ((define content (xexpr-content xe))
          ; XItem.v2 Image -> Image
          (define (deal-with-one item so-far)
            (above/align 'left (render-item item) so-far)))
    (foldr deal-with-one empty-image content)))

; XItem.v2 -> Image
; renders one XItem.v2 as an image
(define (render-item an-item)
  (local ((define content (first (xexpr-content an-item))))
    (bulletize
      (cond
        [(word? content)
         (text (word-text content) SIZE 'black)]
        [else (render-enum content)]))))
```

Figure 128: Refining functions to match refinements of data definitions

The next question is how this change to the data definition affects the rendering functions. Put differently, we need to revise `render-enum1` and `render-item1` so that they can cope with XEnum.v2 and XItem.v2, respectively. Software engineers face these kinds of questions all the time, and it is another situation where the design recipe shines.

Figure 128 shows the complete answer. Since the change is confined to the data definitions for XItem.v2, it should not come as a surprise that the change to the rendering program shows up in the function for rendering items. While `render-item1` does not need to distinguish between different forms of XItem.v1, `render-item` is forced to use a `cond` because XItem.v2 lists two different kinds of items. Given that this data definition is close to one from the real world, the distinguishing characteristic is not something simple—like `'()` vs `cons`—but a specific piece of the given item. If the item's content is a Word, the rendering function proceeds as before. Otherwise, the item contains an enumeration, in which case `render-item` uses `render-enum` to deal with the data, because the data definition for XItem.v2 refers back to XEnum.v2 precisely at this point.

Exercise 373. Figure 128 is missing test cases. Develop test cases for all the functions.

Exercise 374. The data definitions in figure 127 use `list`. Rewrite them so they use `cons`. Then use the recipe to design the rendering functions for XEnum.v2 and XItem.v2 from scratch. You should come up with the same definitions as in figure 128.

Exercise 375. The wrapping of `cond` with

```
(beside/align 'center BT ...)
```

may surprise you. Edit the function definition so that the wrap-around appears once in each clause. Why are you confident that your change works? Which version do you prefer?

Exercise 376. Design a program that counts all `"hello"`s in an instance of XEnum.v2.

Exercise 377. Design a program that replaces all `"hello"`s with `"bye"` in an enumeration.

22.3 Domain-Specific Languages

Engineers routinely build large software systems that require a configuration for specific contexts before they can be run. This configuration task tends to fall to *systems administrators* who must deal with many different software systems. The word "configuration" refers to the data that the main function needs when the program is launched. In a sense a configuration is just an addition argument, though it is usually so complex that program designers prefer a different mechanism for handing it over.

Since software engineers cannot assume that systems administrators know every programming language, they tend to devise simple, special-purpose configuration languages. These special languages are also known as *domain-specific languages* (DSL). Developing these DSLs around a common core, say the well-known XML syntax, simplifies life for systems administrators. They can write small XML "programs" and thus configure the systems they must launch.

> Because configurations abstract a program over various pieces of data, Paul Hudak argued in the 1990s that DSLs are the **ultimate abstractions**, that is, that they generalize the ideas of Abstraction to perfection.

While the construction of a DSL is often considered a task for an advanced programmer, you are actually in a position already to understand, appreciate, and implement a reasonably complex DSL. This section explains how it all works. It first reacquaints you with finite state machines (FSMs). Then it shows how to design, implement, and program a DSL for configuring a system that simulates arbitrary FSMs.

Finite State Machines Remembered The theme of finite state machines is an important one in computing, and this book has presented it several times. Here we reuse the example from Finite State Machines as the component for which we wish to design and implement a configuration DSL.

```
; An FSM is a [List-of 1Transition]
; A 1Transition is a list of two items:
;   (cons FSM-State (cons FSM-State '()))
; An FSM-State is a String that specifies a color

; data examples
(define fsm-traffic
  '(("red" "green") ("green" "yellow") ("yellow" "red")))

; FSM FSM-State -> FSM-State
; matches the keys pressed by a player with the given FSM
(define (simulate state0 transitions)
  (big-bang state0 ; FSM-State
    [to-draw
      (lambda (current)
```

```
        (square 100 "solid" current))]
    [on-key
      (lambda (current key-event)
        (find transitions current))]))

; [X Y] [List-of [List X Y]] X -> Y
; finds the matching Y for the given X in alist
(define (find alist x)
  (local ((define fm (assoc x alist)))
    (if (cons? fm) (second fm) (error "not found"))))
```

Figure 129: Finite state machines, revisited

For convenience, figure 129 presents the entire code again, though reformulated using just lists and using the full power of ISL+. The program consists of two data definitions, one data example, and two function definitions: `simulate` and `find`. Unlike the related programs in preceding chapters, this one represents a transition as a list of two items: the current state and the next one.

The main function, `simulate`, consumes a transition table and an initial state; it then evaluates a `big-bang` expression, which reacts to each key event with a state transition. The states are displayed as colored squares. The `to-draw` and `on-key` clauses are specified with `lambda` expressions that consume the current state, plus the actual key event, and that produce an image or the next state, respectively.

As its signature shows, the auxiliary `find` function is completely independent of the FSM application. It consumes a list of two-item lists and an item, but the actual nature of the items is specified via parameters. In the context of this program, `X` and `Y` represent FSM-States, meaning `find` consumes a transition table together with a state and produces a state. The function body uses the built-in `assoc` function to perform most of the work. Look up the documentation for `assoc` so that you understand why the body of `local` uses an `if` expression.

Exercise 378. Modify the rendering function so that it overlays the name of the state onto the colored square.

Exercise 379. Formulate test cases for `find`.

Exercise 380. Reformulate the data definition for 1Transition so that it is possible to restrict transitions to certain keystrokes. Try to formulate the change so that `find` continues to work without change. What else do you need to change to get the complete program to work? Which part of the design recipe provides the answer(s)? See exercise 229 for the original exercise statement.

Configurations The FSM simulation function uses two arguments, which jointly describe a machine. Rather than teach a potential "customer" how to open an ISL+ program in DrRacket and launch a function of two arguments, the "seller" of `simulate` may wish to supplement this product with a configuration component.

A configuration component consists of two parts. The first one is a widely used simple language that customers use to formulate the initial arguments for a component's main function(s). The second one is a function that translates what customers say into a function call for the main function. For the FSM simulator, we must agree on how we represent finite state machines in XML. By judicious planning, XML as S-expressions presents a series of machine examples that look just right for the task. Recall the final `machine` example in this section:

```
<machine initial="red">
  <action state="red"    next="green" />
  <action state="green"  next="yellow" />
  <action state="yellow" next="red" />
</machine>
```

Compare it to the transition table `fsm-traffic` from figure 129. Also recall the agreed-upon Xexpr representation of this example:

```
(define xm0
  '(machine ((initial "red"))
     (action ((state "red") (next "green")))
     (action ((state "green") (next "yellow")))
     (action ((state "yellow") (next "red")))))
```

What we are still lacking is a general data definition that describes all possible Xexpr representations of FSMs:

```
; An XMachine is a nested list of this shape:
;   `(machine ((initial ,FSM-State)) [List-of X1T])
; An X1T is a nested list of this shape:
;   `(action ((state ,FSM-State) (next ,FSM-State)))
```

Like XEnum.v2, XMachine describes a subset of all Xexpr. Thus, when we design functions that process this new form of data, we may continue to use the generic Xexpr functions to access pieces.

Exercise 381. The definitions of XMachine and X1T use quote, which is highly inappropriate for novice program designers. Rewrite them first to use `list` and then `cons`.

Exercise 382. Formulate an XML configuration for the BW machine, which switches from white to black and back for every key event. Translate the XML configuration into an XMachine representation. See exercise 227 for an implementation of the machine as a program.

Before we dive into the translation part of the configuration problem, let's spell it out:

> **Sample Problem** Design a program that uses an XMachine configuration to run `simulate`.

While this problem is specific to our case, it is easy to imagine a generalization for similar systems, and we encourage you to do so.

The problem statement suggests a complete outline:

```
; XMachine -> FSM-State
; simulates an FSM via the given configuration
(define (simulate-xmachine xm)
  (simulate ... ...))
```

Following the problem statement, our function calls `simulate` with two to-be-determined arguments. What we need to complete the definition are two pieces: an initial state and a transition table. These two pieces are part of `xm`, and we are best off wishing for appropriate functions:

- `xm-state0` extracts the initial state from the given XMachine:

  ```
  (check-expect (xm-state0 xm0) "red")
  ```

- `xm->transitions` translates the embedded list of X1Ts into a list of 1Transitions:

  ```
  (check-expect (xm->transitions xm0) fsm-traffic)
  ```

```
; XMachine -> FSM-State
; interprets the given configuration as a state machine
(define (simulate-xmachine xm)
  (simulate (xm-state0 xm) (xm->transitions xm)))

; XMachine -> FSM-State
; extracts and translates the transition table from xm0

(check-expect (xm-state0 xm0) "red")

(define (xm-state0 xm0)
  (find-attr (xexpr-attr xm0) 'initial))

; XMachine -> [List-of 1Transition]
; extracts the transition table from xm

(check-expect (xm->transitions xm0) fsm-traffic)

(define (xm->transitions xm)
  (local (; X1T -> 1Transition
          (define (xaction->action xa)
            (list (find-attr (xexpr-attr xa) 'state)
                  (find-attr (xexpr-attr xa) 'next))))
    (map xaction->action (xexpr-content xm))))
```

Figure 130: Interpreting a DSL program

Since XMachine is a subset of `Xexpr`, defining `xm-state0` is straightforward. Given that the initial state is specified as an attribute, `xm-state0` extracts the list of attributes using `xexpr-attr` and then retrieves the value of the `'initial` attribute.

Let's then turn to `xm->transitions`, which translates the transitions inside of an `XMachine` configuration into a transition table:

```
; XMachine -> [List-of 1Transition]
; extracts & translates the transition table from xm
(define (xm->transitions xm)
 '())
```

The name of the function prescribes the signature and suggests a purpose statement. Our purpose statement describes a two-step process: (1) extract the `Xexpr` representation of the transitions and (2) translate them into an instance of [List-of 1Transition].

While the extraction part obviously uses `xexpr-content` to get the list, the translation part calls for some more analysis. If you look back to the data definition of XMachine, you see that the content of the Xexpr is a list of X1Ts. The signature tells us that the transition table is a list of 1Transitions. Indeed, it is quite obvious that each item in the former list is translated into one item of the latter, which suggests a use of `map`:

```
(define (xm->transitions xm)
  (local (; X1T -> 1Transition
          (define (xaction->action xa)
            ...))
    (map xaction->action (xexpr-content xm))))
```

As you can see, we follow the design ideas of Using Abstractions, by Example and formulate the function as a `local` whose body uses `map`. Defining `xaction->action` is again just a matter of extracting the appropriate values from an Xexpr.

Figure 130 displays the complete solution. Here the translation from the DSL to a proper function call is as large as the original component. This is not the case for real-world systems; the DSL component tends to be a small fraction of the overall product, which is why the approach is so popular.

Exercise 383. Run the code in figure 130 with the BW Machine configuration from exercise 382.

machine-configuration.xml

```
<machine initial="red">
 <action state="red"    next="green" />
 <action state="green"  next="yellow" />
 <action state="yellow" next="red" />
</machine>
```

Figure 131: A file with a machine configuration

22.4 Reading XML

Systems administrators expect that sophisticated applications read configuration programs from a file or possibly from some place on the web. In ISL+ your programs can retrieve (some) XML information. Figure 132 shows the relevant excerpt from the teachpack. For consistency, the figure uses the suffix `.v3` for its XML representation, including those data definitions for which there is no version 2:

> This section uses *2htdp/batch-io* *2htdp/universe*, and *2htdp/image* libraries.

```
; An Xexpr.v3 is one of:
;  – Symbol
;  – String
;  – Number
;  – (cons Symbol (cons Attribute*.v3 [List-of Xexpr.v3]))
;  – (cons Symbol [List-of Xexpr.v3])
;
; An Attribute*.v3 is a [List-of Attribute.v3].
;
; An Attribute.v3 is a list of two items:
;   (list Symbol String)
```

```
; Any -> Boolean
; is x an Xexpr.v3
; effect displays bad piece if x is not an Xexpr.v3
(define (xexpr? x) ...)

; String -> Xexpr.v3
; produces the first XML element in file f
(define (read-xexpr f) ...)

; String -> Boolean
; #false, if this url returns a '404'; #true otherwise
```

```
(define (url-exists? u) ...)

; String -> [Maybe Xexpr.v3]
; retrieves the first XML (HTML) element from URL u
; #false if (not (url-exists? u))
(define (read-plain-xexpr/web u) ...)

; String -> [Maybe Xexpr.v3]
; retrieves the first XML (HTML) element from URL u
; #false if (not (url-exists? u))
(define (read-xexpr/web u) ...)
```

Figure 132: Reading X-expressions

Assume we have the file in figure 131. If the *2htdp/batch-io* library is required, a program can read the element with `read-plain-xexpr`. The function retrieves the XML element in a format that matches the XMachine data definition. A function for retrieving XML elements from the web is also available in the teachpack. Try this in DrRacket:

```
> (read-plain-xexpr/web
    (string-append
       "http://www.ccs.neu.edu/"
       "home/matthias/"
       "HtDP2e/Files/machine-configuration.xml"))
```

If your computer is connected to the web, this expression retrieves our standard machine configuration.

Reading files or web pages introduces an entirely novel idea into our computational model. As Intermezzo 1: Beginning Student Language explains, a BSL program is evaluated in the same manner in which you evaluate variable expressions in algebra. Function definitions are also treated just like in algebra. Indeed, most algebra courses introduce conditional function definitions, meaning `cond` does not pose any challenges either. Finally, while ISL+ introduces functions as values, the evaluation model remains fundamentally the same.

One essential property of this computational model is that no matter how often you call a function `f` on some argument(s) `a ...`

```
(f a ...)
```

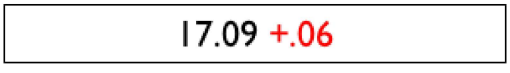
the answer remains the same. The introduction of `read-file`, `read-xexpr`, and their relatives destroys this property, however. The problem is that files and web sites may change over time so that every time a program reads files or web sites it may get a new result.

Consider the idea of looking up the stock price of a company. Point your browser to `google.com/finance` or any other such financial web site and enter the name of your favorite company, say, `Ford`. In response, the site will display the current price of the company's stock and other information—for example, how much the price has changed since the last time it was posted, the current time, and many other facts and ads. The important point is that as you reload this page over the course of a day or a week, some of the information on this web page will change.

An alternative to looking up such company information manually is to write a small program that retrieves such information on a regular basis, say, every 15 seconds. With ISL you can write a world program that performs this task. You would launch it like this:

```
> (stock-alert "Ford")
```

to see a world window that displays an image like the following:



To develop such a program requires skills beyond normal program design. First, you need to investigate how the web site formats its information. In the case of Google's financial service page, an inspection of the web source code shows the following pattern near the top:

```
<meta content="17.09" itemprop="price" />
<meta content="+0.07" itemprop="priceChange" />
<meta content="0.41" itemprop="priceChangePercent" />
<meta content="2013-08-12T16:59:06Z" itemprop="quoteTime" />
<meta content="NYSE real-time data" itemprop="dataSource" />
```

If we had a function that could search an Xexpr.v3 and extract (the representation of XML) `meta` elements with the attribute value `"price"` and `"priceChange"`, the rest of `stock-alert` would be straightforward.

```
(define PREFIX "https://www.google.com/finance?q=")
```

```
(define SIZE 22) ; font size

(define-struct data [price delta])
; A StockWorld is a structure: (make-data String String)

; String -> StockWorld
; retrieves the stock price of co and its change every 15s
(define (stock-alert co)
  (local ((define url (string-append PREFIX co))
          ; [StockWorld -> StockWorld]
          (define (retrieve-stock-data __w)
            (local ((define x (read-xexpr/web url)))
              (make-data (get x "price")
                         (get x "priceChange"))))
          ; StockWorld -> Image
          (define (render-stock-data w)
            (local (; [StockWorld -> String] -> Image
                    (define (word sel col)
                      (text (sel w) SIZE col)))
              (overlay (beside (word data-price 'black)
                               (text "  " SIZE 'white)
                               (word data-delta 'red))
                       (rectangle 300 35 'solid 'white)))))
    (big-bang (retrieve-stock-data 'no-use)
      [on-tick retrieve-stock-data 15]
      [to-draw render-stock-data])))
```

Figure 133: Web data as an event

Figure 133 displays the core of the program. The design of `get` is left to the exercises because its workings are all about intertwined data.

As the figure shows, the main function defines two local ones: a clock-tick handler and a rendering function. The `big-bang` specification requests that the clock tick every 15 seconds. When the clock ticks, ISL+ applies `retrieve-stock-data` to the current world, which it ignores. Instead, the function visits the web site via `read-xexpr/web` and extracts the appropriate information with `get`. Thus, the new world is created from newly available information on the web, not some local data.

Exercise 384. Figure 133 mentions `read-xexpr/web`. See figure 132 for its signature and purpose statement and then read its documentation to determine the difference to its "plain" relative.

Figure 133 is also missing several important pieces, in particular the interpretation of `data` and purpose statements for all the locally defined functions. Formulate the missing pieces so that you get to understand the program.

Exercise 385. Look up the current stock price for your favorite company at Google's financial service page. If you don't favor a company, pick Ford. Then save the source code of the page as a file in your working directory. Use `read-xexpr` in DrRacket to view the source as an Xexpr.v3.

Exercise 386. Here is the `get` function:

```
; Xexpr.v3 String -> String
; retrieves the value of the "content" attribute
; from a 'meta element that has attribute "itemprop"
; with value s
(check-expect
  (get '(meta ((content "+1") (itemprop "F"))) "F")
  "+1")

(define (get x s)
  (local ((define result (get-xexpr x s)))
    (if (string? result)
        result
        (error "not found"))))
```

It assumes the existence of `get-xexpr`, a function that searches an arbitrary Xexpr.v3 for the desired attribute and produces [Maybe String].

Formulate test cases that look for other values than `"F"` and that force `get` to signal an error.

Design `get-xexpr`. Derive functional examples for this function from those for `get`. Generalize these examples so that you are confident `get-xexpr` can traverse an arbitrary Xexpr.v3. Finally, formulate a test that uses the web data saved in

exercise 385.

23 Simultaneous Processing

Some functions have to consume two arguments that belong to classes with nontrivial data definitions. How to design such functions depends on the relationship between the arguments. First, one of the arguments may have to be treated as if it were atomic. Second, it is possible that the function must process the two arguments in lockstep. Finally, the function may process the given data in accordance to all possible cases. This chapter illustrates the three cases with examples and provides an augmented design recipe. The last section discusses the equality of compound data.

23.1 Processing Two Lists Simultaneously: Case 1

Consider the following signature, purpose statement, and header:

```
; [List-of Number] [List-of Number] -> [List-of Number]
; replaces the final '() in front with end
(define (replace-eol-with front end)
  front)
```

The signature says that the function consumes two lists. Let's see how the design recipe works in this case.

We start by working through examples. If the first argument is `'()`, `replace-eol-with` must produce the second one, no matter what it is:

```
(check-expect (replace-eol-with '() '(a b)) '(a b))
```

In contrast, if the first argument is not `'()`, the purpose statement requires that we replace `'()` at the end of `front` with `end`:

```
(check-expect (replace-eol-with (cons 1 '()) '(a))
              (cons 1 '(a)))
(check-expect (replace-eol-with
                (cons 2 (cons 1 '())) '(a))
              (cons 2 (cons 1 '(a))))
```

The purpose statement and the examples suggest that as long as the second argument is a list, the function does not need to know anything about it. By implication, its template should be that of a list-processing function with respect to the first argument:

```
(define (replace-eol-with front end)
  (cond
    [(empty? front) ...]
    [else
     (... (first front) ...
      ... (replace-eol-with (rest front) end) ...)]))
```

Let's fill the gaps in the template following the fifth step of the design recipe. If `front` is `'()`, `replace-eol-with` produces `end`. If `front` is not `'()`, we must recall what the template expressions compute:

- `(first front)` evaluates to the first item on the list, and
- `(replace-eol-with (rest front) end)` replaces the final `'()` in `(rest front)` with `end`.

Stop! Use the table method to understand what these bullets mean for the running example.

From here it is a small step to the complete definition:

```
(define (replace-eol-with front end)
  (cond
    [(empty? front) end]
    [else
     (cons (first front)
           (replace-eol-with (rest front) end))]))
```

Exercise 387. Design `cross`. The function consumes a list of symbols and a list of numbers and produces all possible ordered pairs of symbols and numbers. That is, when given `'(a b c)` and `'(1 2)`, the expected result is `'((a 1) (a 2) (b 1) (b 2) (c 1) (c 2))`.

23.2 Processing Two Lists Simultaneously: Case 2

Functions that Produce Lists presents the function `wages*`, which computes the weekly wages of some workers given their work hours. It consumes a list of numbers, which represents the hours worked per week, and produces a list of numbers, which are the corresponding weekly wages. While the problem assumes that all employees received the same pay rate, even a small company pays its workers differentiated wages.

Here we look at a slightly more realistic version. The function now consumes **two** lists: the list of hours worked and the list of corresponding hourly wages. We translate this revised problem into a revised header:

```
; [List-of Number] [List-of Number] -> [List-of Number]
; multiplies the corresponding items on
; hours and wages/h
; assume the two lists are of equal length
(define (wages*.v2 hours wages/h)
  '())
```

Making up examples is straightforward:

```
(check-expect (wages*.v2 '() '()) '())
(check-expect (wages*.v2 (list 5.65) (list 40))
              (list 226.0))
(check-expect (wages*.v2 '(5.65 8.75) '(40.0 30.0))
              '(226.0 262.5))
```

As required, all three examples use lists of equal length.

The assumption concerning the inputs can also be exploited for the development of the template. More concretely, the condition says that `(empty? hours)` is true when `(empty? wages/h)` is true, and furthermore, `(cons? hours)` is true when `(cons? wages/h)` is true. It is thus acceptable to use a template for one of the two lists:

```
(define (wages*.v2 hours wages/h)
  (cond
    [(empty? hours) ...]
    [else
     (... (first hours)
      ... (first wages/h) ...
      ... (wages*.v2 (rest hours) (rest wages/h)))]))
```

In the first `cond` clause, both `hours` and `wages/h` are `'()`. Hence no selector expressions are needed. In the second clause, both `hours` and `wages/h` are `constructed` lists, which means we need four selector expressions. Finally, because the last two are lists of equal length, they make up a natural candidate for the natural recursion of `wages*.v2`.

The only unusual aspect of this template is that the recursive application consists of two expressions, both selector expressions for the two arguments. But, this idea directly follows from the assumption.

From here, it is a short step to a complete function definition:

```
(define (wages*.v2 hours wages/h)
  (cond
    [(empty? hours) '()]
    [else
     (cons
       (weekly-wage (first hours) (first wages/h))
       (wages*.v2 (rest hours) (rest wages/h)))]))
```

The first example implies that the answer for the first `cond` clause is `'()`. In the second one, we have three values available:

1. `(first hours)`, which represents the first number of weekly hours;
2. `(first wages/h)`, which is the first pay rate; and
3. `(wages*.v2 (rest hours) (rest wages/h))`, which, according to the purpose statement, computes the list of weekly wages for the remainders of the two lists.

Now we just need to combine these values to get the final answer. As suggested by the examples, we must compute the weekly wage for the first employee and `construct` a list from that wage and the rest of the wages:

```
(cons (weekly-wage (first hours) (first wages/h))
      (wages*.v2 (rest hours) (rest wages/h)))
```

The auxiliary function `weekly-wage` uses the number of hours worked and the pay rate to compute the weekly wage for one worker:

```
; Number Number -> Number
; computes the weekly wage from pay-rate and hours
(define (weekly-wage pay-rate hours)
  (* pay-rate hours))
```

Stop! Which function do you need to use if you wish to compute the wages for one worker? Which function do you need to change if you wish to deal with income taxes?

Exercise 388. In the real world, `wages*.v2` consumes lists of employee structures and lists of work records. An employee structure contains an employee's name, social security number, and pay rate. A work record also contains an employee's name and the number of hours worked in a week. The result is a list of structures that contain the name of the employee and the weekly wage.

Modify the program in this section so that it works on these realistic versions of data. Provide the necessary structure type definitions and data definitions. Use the design recipe to guide the modification process.

Exercise 389. Design the `zip` function, which consumes a list of names, represented as strings, and a list of phone numbers, also strings. It combines those equally long lists into a list of phone records:

```
(define-struct phone-record [name number])
; A PhoneRecord is a structure:
;   (make-phone-record String String)
```

Assume that the corresponding list items belong to the same person.

23.3 Processing Two Lists Simultaneously: Case 3

Here is a third type of problem:

> **Sample Problem** Given a list of symbols `los` and a natural number `n`, the function `list-pick` extracts the `n`th symbol from `los`; if there is no such symbol, it signals an error.

The question is how well the recipe works for the design of `list-pick`.

While the data definition for a list of symbols is fairly familiar by now, recall the class of natural numbers from Natural Numbers:

```
; N is one of:
; – 0
; – (add1 N)
```

Now we can proceed to the second step:

```
; [List-of Symbol] N -> Symbol
; extracts the nth symbol from l;
; signals an error if there is no such symbol
(define (list-pick l n)
  'a)
```

Both lists of symbols and natural numbers are classes with complex data definitions. This combination makes the problem nonstandard, meaning we must pay attention to every detail for every step of the design recipe.

At this point, we usually pick some input examples and figure out what the desired output is. We start with inputs for which the function has to work flawlessly: `'(a b c)` and `2`. For a list of three symbols and the index `2`, `list-pick` must return a symbol. The question is whether it is `'b` or `'c`. In grade school, you would have counted `1`, `2`, and picked `'b` without a first thought. But this is computer science, not grade school. Here people start counting from `0`, meaning that `'c` is an equally appropriate choice. And indeed, this is the choice we use:

```
(check-expect (list-pick '(a b c) 2) 'c)
```

Now that we have eliminated this fine point of `list-pick`, let's look at the actual problem, the choice of inputs. The goal of the example step is to cover the input space as much as possible. We do so by picking one input per clause in the description of complex forms of data. Here this procedure suggests we pick at least two elements from each class because each data definition has two clauses. We choose `'()` and `(cons 'a '())` for the first argument, and `0` and `3` for the latter. Two choices per argument means four examples total; after all, there is no immediately obvious connection between the two arguments and no restriction in the signature.

As it turns out, only one of these pairings produces a proper result; the remaining ones choose a position that does not exist because the list doesn't contain enough symbols:

```
(check-error (list-pick '() 0) "list too short")
(check-expect (list-pick (cons 'a '()) 0) 'a)
(check-error (list-pick '() 3) "list too short")
```

The function is expected to signal an error, and we pick our favorite message here.

Stop! Put these fragments into DrRacket's definitions area and run the partial program.

The discussion on examples indicates that there are indeed four independent cases that we must inspect for the design of the function. One way to discover these cases is to arrange the conditions for each of the clauses into a two-dimensional table:

	`(empty? l)`	`(cons? l)`
`(= n 0)`		
`(> n 0)`		

The horizontal dimension of the table lists those questions that `list-pick` must ask about lists; the vertical dimension lists the questions about natural numbers. By this arrangement, we naturally get four squares, where each represents the case when both the conditions on the horizontal and the vertical axis are true.

Our table suggests that the `cond` for the function template has four clauses. We can figure out the appropriate condition for each of these clauses by `and`-ing the horizontal and vertical condition for each box in the table:

	`(empty? l)`	`(cons? l)`
`(= n 0)`	`(and (empty? l) (= n 0))`	`(and (cons? l) (= n 0))`
`(> n 0)`	`(and (empty? l) (> n 0))`	`(and (cons? l) (> n 0))`

The `cond` outline of the template is merely a translation of this table into a conditional:

```
(define (list-pick l n)
  (cond
    [(and (= n 0) (empty? l)) ...]
    [(and (> n 0) (empty? l)) ...]
    [(and (= n 0) (cons? l)) ...]
    [(and (> n 0) (cons? l)) ...]))
```

As always, the `cond` expression allows us to distinguish the four possibilities and to focus on each individually as we add selector expressions to each `cond` clause:

```
(define (list-pick l n)
  (cond
    [(and (= n 0) (empty? l))
     ...]
    [(and (> n 0) (empty? l))
     (... (sub1 n) ...)]
    [(and (= n 0) (cons? l))
     (... (first l) ... (rest l)...)]
    [(and (> n 0) (cons? l))
     (... (sub1 n) ... (first l) ... (rest l) ...)]))
```

The first argument, `l`, is a list, and a template's `cond` clause for non-empty lists contains two selector expressions. The second argument, `n`, belongs to N, and the template's `cond` clause for non-`0` numbers needs only one selector expression. In those cases where either `(empty? l)` or `(= n 0)` holds, the respective argument is atomic and there is no need for a corresponding selector expression.

The final step of the template construction demands that we annotate the template with recursions where the results of selector expressions belong to the same class as the inputs. For this first example, we focus on the last `cond` clause, which contains selector expressions for both arguments. It is, however, unclear how to form the natural recursions. If we disregard the purpose of the function, there are three possible recursions:

1. `(list-pick (rest l) (sub1 n))`

2. `(list-pick l (sub1 n))`

3. `(list-pick (rest l) n)`

Each one represents a feasible combination of the available expressions. Since we cannot know which one matters or whether all three matter, we move on to the next development stage.

```
; [List-of Symbol] N -> Symbol
; extracts the nth symbol from l;
; signals an error if there is no such symbol
(define (list-pick l n)
  (cond
    [(and (= n 0) (empty? l))
     (error 'list-pick "list too short")]
    [(and (> n 0) (empty? l))
     (error 'list-pick "list too short")]
    [(and (= n 0) (cons? l)) (first l)]
    [(and (> n 0) (cons? l)) (list-pick (rest l) (sub1 n))]))
```

Figure 134: Indexing into a list

Following the design recipe for the fifth step, let's analyze each `cond` clause in the template and decide what a proper answer is:

1. If `(and (= n 0) (empty? l))` holds, `list-pick` must pick the first symbol from an empty list, which is impossible. The answer must be an error signal.

2. If `(and (> n 0) (empty? l))` holds, `list-pick` is again asked to pick a symbol from an empty list.

3. If `(and (= n 0) (cons? l))` holds, `list-pick` is supposed to produce the first symbol from `l`. The selector expression `(first l)` is the answer.

4. If `(and (> n 0) (cons? l))` holds, we must analyze what the available expressions compute. As we have seen, it is a good idea to work through an existing example for this step. We pick a shortened variant of the first example:

   ```
   (check-expect (list-pick '(a b) 1) 'b)
   ```

   Here is what the three natural recursions compute with these values:

   a. `(list-pick '(b) 0)` produces `'b`;

   b. `(list-pick '(a b) 0)` evaluates to `'a`, the wrong answer;

   c. and `(list-pick '(b) 1)` signals an error.

   From this, we conclude that `(list-pick (rest l) (sub1 n))` computes the desired answer in the last `cond` clause.

Exercise 390. Design the function `tree-pick`. The function consumes a tree of symbols and a list of directions:

```
(define-struct branch [left right])

; A TOS is one of:
; – Symbol
; – (make-branch TOS TOS)

; A Direction is one of:
; – 'left
; – 'right

; A list of Directions is also called a path.
```

Clearly a Direction tells the function whether to choose the left or the right branch in a nonsymbolic tree. What is the result of the `tree-pick` function? Don't forget to formulate a full signature. The function signals an error when given a symbol and a non-empty path.

23.4 Function Simplification

The `list-pick` function in figure 134 is far more complicated than necessary. The first two `cond` clauses signal an error. That is, if either

```
(and (= n 0) (empty? alos))
```

or

```
(and (> n 0) (empty? alos))
```

holds, the answer is an error. We can translate this observation into code:

```
(define (list-pick alos n)
```

```
(cond
  [(or (and (= n 0) (empty? alos))
       (and (> n 0) (empty? alos)))
   (error 'list-pick "list too short")]
  [(and (= n 0) (cons? alos)) (first alos)]
  [(and (> n 0) (cons? alos))
   (list-pick (rest alos) (sub1 n))]))
```

To simplify this function even more, we need to get acquainted with algebraic laws concerning Booleans:

These equations are known as de Morgan's laws.

```
(or (and bexp1 a-bexp)   ==  (and (or bexp1 bexp2)
    (and bexp2 a-bexp))           a-bexp)
```

A similar law applies when the sub-expressions of the `and`s are swapped. Applying these laws to `list-pick` yields this:

```
(define (list-pick n alos)
  (cond
    [(and (or (= n 0) (> n 0)) (empty? alos))
     (error 'list-pick "list too short")]
    [(and (= n 0) (cons? alos)) (first alos)]
    [(and (> n 0) (cons? alos))
     (list-pick (rest alos) (sub1 n))]))
```

Now consider `(or (= n 0) (> n 0))`. It is always `#true` because `n` belongs to N. Since `(and #true (empty? alos))` is equivalent to `(empty? alos)`, we can rewrite the function again:

```
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(and (= n 0) (cons? alos)) (first alos)]
    [(and (> n 0) (cons? alos))
     (list-pick (rest alos) (sub1 n))]))
```

This last definition is already significantly simpler than the definition in figure 134, but we can do even better than this. Compare the first condition in the latest version of `list-pick` with the second and third. Since the first `cond` clause filters out all those cases when `alos` is empty, `(cons? alos)` in the last two clauses is always going to evaluate to `#true`. If we replace the condition with `#true` and simplify the `and` expressions again, we get a three-line version of `list-pick`

```
; list-pick: [List-of Symbol] N[>= 0] -> Symbol
; determines the nth symbol from alos, counting from 0;
; signals an error if there is no nth symbol
(define (list-pick alos n)
  (cond
    [(empty? alos) (error 'list-pick "list too short")]
    [(= n 0) (first alos)]
    [(> n 0) (list-pick (rest alos) (sub1 n))]))
```

Figure 135: Indexing into a list, simplified

Figure 135 displays this simplified version of `list-pick`. While it is far simpler than the original, it is important to understand that we designed the original in a systematic manner and that we were able to transform the first into the second with well-established algebraic laws. We can therefore trust this simple version. If we try to find the simple versions of functions directly, we sooner or later fail to take care of a case in our analysis, and we are guaranteed to produce flawed programs.

Exercise 391. Design `replace-eol-with` using the strategy of Processing Two Lists Simultaneously: Case 3. Start from the tests. Simplify the result systematically.

Exercise 392. Simplify the function `tree-pick` from exercise 390.

23.5 Designing Functions that Consume Two Complex Inputs

The proper approach to designing functions of two (or more) complex arguments is to follow the general recipe. You must conduct a data analysis and define the relevant classes of data. If the use of parametric definitions such as List-of and short-hand examples such as `'(1 b &)` confuses you, expand them so that the constructors become explicit. Next you need a

function signature and purpose. At this point, you can think ahead and decide which of the following three situations you are facing:

1. If one of the parameters plays a dominant role, think of the other as an atomic piece of data as far as the function is concerned.

2. In some cases the parameters range over the same class of values and must have the same size. For example, two lists must have the same length, or two web pages must have the same length and where one of them contains an embedded page, the other one does, too. If the two parameters have this equal status and the purpose suggests that they are processed in a synchronized manner, you choose one parameter, organize the function around it, and traverse the other in a parallel manner.

3. If there is no obvious connection between the two parameters, you must analyze all possible cases with examples. Then use this analysis to develop the template, especially the recursive parts.

Once you decide that a function falls into the third category, develop a two-dimensional table to make sure that no case falls through the cracks. Let's use a nontrivial pair of data definitions to explain this idea again:

```
; An LOD is one of:       ; A TID is one of:
; – '()                   ; – Symbol
; – (cons Direction LOD)  ; – (make-binary TID TID)
                          ; – (make-with TID Symbol TID)
```

The left data definition is the usual list definition; the right one is a three-clause variant of TOS. It uses two structure type definitions:

```
(define-struct with [lft info rght])
(define-struct binary [lft rght])
```

Assuming the function consumes an LOD and a TID, the table that you should come up with has this shape:

	(empty? l)	(cons? l)
(symbol? t)		
(binary? t)		
(with? t)		

Along the horizontal direction we enumerate the conditions that recognize the sub-classes for the first parameter, here LOD, and along the vertical direction we enumerate the conditions for the second parameter, TID.

The table guides the development of both the function examples and the function template. As explained, the examples must cover all possible cases; that is, there must be at least one example for each cell in the table. Similarly, the template must have one `cond` clause per cell; its condition combines the horizontal and the vertical conditions in an `and` expression. Each `cond` clause, in turn, must contain all feasible selector expressions for both parameters. If one of the parameters is atomic, there is no need for a selector expression. Finally, you need to be aware of the feasible natural recursions. In general, all possible combinations of selector expressions (and optionally, atomic arguments) are candidates for a natural recursion. Because we can't know which ones are necessary and which ones aren't, we keep them in mind for the coding step.

In summary, the design of multiparameter functions is just a variation on the old design-recipe theme. The key idea is to translate the data definitions into a table that shows all feasible and interesting combinations. The development of function examples and the template exploit the table as much as possible.

23.6 Finger Exercises: Two Inputs

Exercise 393. Figure 62 presents two data definitions for finite sets. Design the `union` function for the representation of finite sets of your choice. It consumes two sets and produces one that contains the elements of both.

Design `intersect` for the same set representation. It consumes two sets and produces the set of exactly those elements that occur in both.

Exercise 394. Design `merge`. The function consumes two lists of numbers, sorted in ascending order. It produces a single sorted list of numbers that contains all the numbers on both inputs lists. A number occurs in the output as many times as it occurs on the two input lists together.

Exercise 395. Design `take`. It consumes a list `l` and a natural number `n`. It produces the first `n` items from `l` or all of `l` if it is too short.

Design `drop`. It consumes a list `l` and a natural number `n`. Its result is `l` with the first `n` items removed or just `'()` if `l` is too short.

```
; An HM-Word is a [List-of Letter or "_"]
```

```
; interpretation "_" represents a letter to be guessed

; HM-Word N -> String
; runs a simplistic hangman game, produces the current state
(define (play the-pick time-limit)
  (local ((define the-word  (explode the-pick))
          (define the-guess (make-list (length the-word) "_"))
          ; HM-Word -> HM-Word
          (define (do-nothing s) s)
          ; HM-Word KeyEvent -> HM-Word
          (define (checked-compare current-status ke)
            (if (member? ke LETTERS)
                (compare-word the-word current-status ke)
                current-status)))
    (implode
     (big-bang the-guess ; HM-Word
       [to-draw render-word]
       [on-tick do-nothing 1 time-limit]
       [on-key  checked-compare]))))

; HM-Word -> Image
(define (render-word w)
  (text (implode w) 22 "black"))
```

Figure 136: A simple hangman game

Exercise 396. Hangman is a well-known guessing game. One player picks a word, the other player gets told how many letters the word contains. The latter picks a letter and asks the first player whether and where this letter occurs in the chosen word. The game is over after an agreed-upon time or number of rounds.

Figure 136 presents the essence of a time-limited version of the game. See Local Definitions Add Expressive Power for why `checked-compare` is defined locally.

The goal of this exercise is to design `compare-word`, the central function. It consumes the word to be guessed, a word `s` that represents how much/little the guessing player has discovered, and the current guess. The function produces `s` with all `"_"` where the guess revealed a letter.

Once you have designed the function, run the program like this:

```
(define LOCATION "/usr/share/dict/words") ; on OS X
(define AS-LIST (read-lines LOCATION))
(define SIZE (length AS-LIST))

(play (list-ref AS-LIST (random SIZE)) 10)
```

See figure 74 for an explanation. Enjoy and refine as desired!

Exercise 397. In a factory, employees punch time cards as they arrive in the morning and leave in the evening. Electronic time cards contain an employee number and record the number of hours worked per week. Employee records always contain the name of the employee, an employee number, and a pay rate.

Design `wages*.v3`. The function consumes a list of employee records and a list of time-card records. It produces a list of wage records, which contain the name and weekly wage of an employee. The function signals an error if it cannot find an employee record for a time card or vice versa.

Assumption There is at most one time card per employee number.

Exercise 398. A linear combination is the sum of many linear terms, that is, products of variables and numbers. The latter are called coefficients in this context. Here are some examples:

$5 \cdot x$ $\qquad$ $5 \cdot x + 17 \cdot y$ $\qquad$ $5 \cdot x + 17 \cdot y + 3 \cdot z$

In all examples, the coefficient of x is 5, that of y is 17, and the one for z is 3.

If we are given values for variables, we can determine the value of a polynomial. For example, if $x = 10$, the value of $5 \cdot x$ is 50; if $x = 10$ and $y = 1$, the value of $5 \cdot x + 17 \cdot y$ is 67; and if $x = 10$, $y = 1$, and $z = 2$, the value of $5 \cdot x + 17 \cdot y + 3 \cdot z$ is 73.

There are many different representations of linear combinations. We could, for example, represent them with functions. An alternative representation is a list of its coefficients. The above combinations would be represented as:

```
(list 5)
(list 5 17)
```

```
(list 5 17 3)
```

This choice of representation assumes a fixed order of variables.

Design `value`. The function consumes two equally long lists: a linear combination and a list of variable values. It produces the value of the combination for these values.

Exercise 399. Louise, Jane, Laura, Dana, and Mary decide to run a lottery that assigns one gift recipient to each of them. Since Jane is a developer, they ask her to write a program that performs this task in an impartial manner. Of course, the program must not assign any of the sisters to herself.

Here is the core of Jane's program:

```
; [List-of String] -> [List-of String]
; picks a random non-identity arrangement of names
(define (gift-pick names)
  (random-pick
    (non-same names (arrangements names))))

; [List-of String] -> [List-of [List-of String]]
; returns all possible permutations of names
; see exercise 213
(define (arrangements names)
  ...)
```

It consumes a list of names and randomly picks one of those permutations that do not agree with the original list at any place.

Your task is to design two auxiliary functions:

```
; [NEList-of X] -> X
; returns a random item from the list
(define (random-pick l)
  (first l))

; [List-of String] [List-of [List-of String]]
; ->
; [List-of [List-of String]]
; produces the list of those lists in ll that do
; not agree with names at any place
(define (non-same names ll)
  ll)
```

Recall that `random` picks a random number; see exercise 99.

Exercise 400. Design the function `DNAprefix`. The function takes two arguments, both lists of `'a`, `'c`, `'g`, and `'t`, symbols that occur in DNA descriptions. The first list is called a pattern, the second one a search string. The function returns `#true` if the pattern is identical to the initial part of the search string; otherwise it returns `#false`.

Also design `DNAdelta`. This function is like `DNAprefix` but returns the first item in the search string beyond the pattern. If the lists are identical and there is no DNA letter beyond the pattern, the function signals an error. If the pattern does not match the beginning of the search string, it returns `#false`. The function must not traverse either of the lists more than once.

Can `DNAprefix` or `DNAdelta` be simplified?

Exercise 401. Design `sexp=?`, a function that determines whether two S-expressions are equal. For convenience, here is the data definition in condensed form:

```
; An S-expr (S-expression) is one of:
; – Atom
; – [List-of S-expr]
;
; An Atom is one of:
; – Number
; – String
; – Symbol
```

Whenever you use `check-expect`, it uses a function like `sexp=?` to check whether the two arbitrary values are equal. If not, the check fails and `check-expect` reports it as such.

Exercise 402. Reread exercise 354. Explain the reasoning behind our hint to think of the given expression as an atomic value at first.

23.7 Project: Database

Many software applications use a database to keep track of data. Roughly speaking, a database is a table that comes with an explicitly stated organization rule. The former is the *content*; the latter is called a *schema*. Figure 137 shows two examples. Each table consists of two parts: the schema above the line and the content below.

> This section pulls together knowledge from all four parts of the book.

Let's focus on the table on the left. It has three columns and four rows. Each column comes with a two-part rule:

1. the rule in the left-most column says that the label of the column is "Name" and that every piece of data in this column is a String;
2. the middle column is labeled "Age" and contains Integers; and
3. the label of the right-most one is "Present"; its values are Boolean.

Stop! Explain the table on the right in the same way.

Name	Age	Present
String	Integer	Boolean
"Alice"	35	#true
"Bob"	25	#false
"Carol"	30	#true
"Dave"	32	#false

Present	Description
Boolean	String
#true	"presence"
#false	"absence"

Figure 137: Databases as tables

Computer scientists think of these tables as *relations*. The schema introduces terminology to refer to columns of a relation and to individual cells in a row. Each row relates a fixed number of values; the collection of all rows makes up the entire relation. In this terminology, the first row of the left table in figure 137 relates `"Alice"` to `35` and `#true`. Furthermore, the first cell of a row is called the "Name" cell, the second the "Age" cell, and the third one the "Present" cell.

In this section, we represent databases via structures and lists:

```
(define-struct db [schema content])
; A DB is a structure:
;   (make-db Schema Content)

; A Schema is a [List-of Spec]
; A Spec is a [List Label Predicate]
; A Label is a String
; A Predicate is a [Any -> Boolean]

; A (piece of) Content is a [List-of Row]
; A Row is a [List-of Cell]
; A Cell is Any
; constraint cells do not contain functions

; integrity constraint In (make-db sch con),
; for every row in con,
; (I1) its length is the same as sch's, and
; (I2) its ith Cell satisfies the ith Predicate in sch
```

Stop! Translate the databases from figure 137 into the chosen data representation. Note that the content of the tables already uses ISL+ data.

```
(define school-schema
  `(("Name"    ,string?)
    ("Age"     ,integer?)
    ("Present" ,boolean?)))

(define school-content
  `(("Alice" 35 #true)
```

```
(define presence-schema
  `(("Present"     ,boolean?)
    ("Description" ,string?)))


(define presence-content
  `((#true  "presence")
```

```
    ("Bob"   25 #false)
    ("Carol" 30 #true)
    ("Dave"  32 #false)))

(define school-db
  (make-db school-schema
           school-content))
```

```
   (#false "absence")))


(define presence-db
  (make-db presence-schema
           presence-content))
```

Figure 138: Databases as ISL+ data

Figure 138 shows how to represent the two tables in figure 137 as DBs. Its left-hand side represents the schema, the content, and the database from the left-hand side table in figure 137; its right part corresponds to the right-hand side table. For succinctness, the examples use the `quasiquote` and `unquote` notation. Recall that it allows the inclusion of a value such as `boolean?` in an otherwise quoted list. If you feel uncomfortable with this notation, reformulate these examples with `list`.

Exercise 403. A Spec combines one Label and one Predicate into a list. While this choice is perfectly acceptable for a mature programmer, it violates our guideline of using a structure type when the represented information consists of a fixed number of pieces.

Here is an alternative data representation:

```
(define-struct spec [label predicate])
; Spec is a structure: (make-spec Label Predicate)
```

Use this alternative definition to represent the databases from figure 137.

Integrity Checking The use of databases critically relies on their integrity. Here "integrity" refers to the constraints (I1) and (I2) from the data definition. Checking database integrity is clearly a task for a function:

```
; DB -> Boolean
; do all rows in db satisfy (I1) and (I2)

(check-expect (integrity-check school-db) #true)
(check-expect (integrity-check presence-db) #true)

(define (integrity-check db)
  #false)
```

The wording of the two constraints suggests that some function has to produce `#true` for every row in the content of the given database. Expressing this idea in code calls for a use of `andmap` on the content of `db`:

```
(define (integrity-check db)
  (local (; Row -> Boolean
          (define (row-integrity-check row)
            ...))
    (andmap row-integrity-check (db-content db))))
```

Following the design recipe for the use of existing abstractions, the template introduces the auxiliary function via a `local` definition.

The design of `row-integrity-check` starts from this:

```
; Row -> Boolean
; does row satisfy (I1) and (I2)
(define (row-integrity-check row)
  #false)
```

As always, the goal of formulating a purpose statement is to understand the problem. Here it says that the function checks **two** conditions. When two tasks are involved, our design guidelines call for functions and the combination of their results:

```
(and (length-of-row-check row)
     (check-every-cell row))
```

Add these functions to your wish list; their names convey their purpose.

Before we design these functions, we must contemplate whether we can compose existing primitive operations to compute the desired value. For example, we know that `(length row)` counts how many cells are in `row`. Pushing a bit more in this direction, we clearly want

```
(= (length row) (length (db-schema db)))
```

This condition checks that the length of `row` is equal to that of `db`'s schema.

Similarly, `check-every-cell` calls for checking that some function produces `#true` for every cell in the row. Once again, it looks like `andmap` might be called for:

```
(andmap cell-integrity-check row)
```

The purpose of `cell-integrity-check` is obviously to check constraint (I2), that is,

> whether "the *i*th Cell satisfies the *i*th Predicate in `db`'s schema."

And now we are stuck because this purpose statement refers to the relative position of the given cell in `row`. The point of `andmap` is, however, to apply `cell-integrity-check` to every cell **uniformly**.

When you are stuck, you must work through examples. For auxiliary or `local` functions, it's best to derive these examples from the ones for the main function. The first example for `integrity-check` asserts that `school-content` satisfies the integrity constraints. Clearly all rows in `school-content` have the same length as `school-schema`. The question is why a row such as

```
(list "Alice" 35 #true)
```

satisfies the predicates in the corresponding schema:

```
(list (list "Name"    string?)
      (list "Age"     integer?)
      (list "Present" boolean?))
```

The answer is that all three applications of the three predicates to the respective cells yields true:

```
> (string? "Alice")
#true
> (integer? 35)
#true
> (boolean? #true)
#true
```

From here, it is just a short step to see that the function must process these two lists—`db`'s schema and the given row—in parallel.

Exercise 404. Design the function `andmap2`. It consumes a function `f` from two values to Boolean and two equally long lists. Its result is also a Boolean. Specifically, it applies `f` to pairs of corresponding values from the two lists, and if `f` always produces `#true`, `andmap2` produces `#true`, too. Otherwise, `andmap2` produces `#false`. In short, `andmap2` is like `andmap` but for two lists.

Stop! Solve exercise 404 before reading on.

If we had `andmap2` in ISL+, checking the second condition on `row` would be straightforward:

```
(andmap2 (lambda (s c) [(second s) c])
         (db-schema db)
         row)
```

The given function consumes a Spec `s` from `db`'s schema, extracts the predicate in the second position, and applies it to the given Cell `c`. Whatever the predicate returns is the result of the `lambda` function.

Stop again! Explain `[(second s) c]`.

As it turns out, `andmap` in ISL+ is already like `andmap2`:

```
(define (integrity-check db)
  (local (; Row -> Boolean
          ; does row satisfy (I1) and (I2)
          (define (row-integrity-check row)
            (and (= (length row)
                    (length (db-schema db)))
                 (andmap (lambda (s c) [(second s) c])
                         (db-schema db)
                         row))))
    (andmap row-integrity-check (db-content db))))
```

Stop a last time! Develop a test for which `integrity-check` must fail.

Note on Expression Hoisting Our definition of `integrity-check` suffers from several problems, some visible, some invisible. Clearly, the function extracts `db`'s schema twice. With the existing `local` definition it is possible to introduce a definition and avoid this duplication:

```
(define (integrity-check.v2 db)
  (local ((define schema (db-schema db))
          ; Row -> Boolean
          ; does row satisfy (I1) and (I2)
          (define (row-integrity-check row)
            (and (= (length row) (length schema))
                 (andmap (lambda (s c) [(second s) c])
                         schema
                         row))))
    (andmap row-integrity-check (db-content db))))
```

We know from Local Definitions that lifting such an expression may shorten the time needed to run the integrity check. Like the definition of `inf` in figure 100, the original version of `integrity-check` extracts the schema from `db` for every single row, even though it obviously stays the same.

```
(define (integrity-check.v3 db)
  (local ((define schema  (db-schema db))
          (define content (db-content db))
          (define width   (length schema))
          ; Row -> Boolean
          ; does row satisfy (I1) and (I2)
          (define (row-integrity-check row)
            (and (= (length row) width)
                 (andmap (lambda (s c) [(second s) c])
                         schema
                         row))))
    (andmap row-integrity-check content)))
```

Figure 139: The result of systematic expression hoisting

Terminology Computer scientists speak of "*hoisting* an expression." In a similar vein, the `row-integrity-check` function determines the length of `db`'s schema every single time it is called. The result is always the same. Hence, if we are interested in improving the performance of this function, we can use a `local` definition to name the `width` of the database content once and for all. Figure 139 displays the result of hoisting `(length schema)` out of the `row-integrity-check`. For readability, this final definition also names the `content` field of `db`. **End**

Projections and Selections Programs need to extract data from databases. One kind of extraction is to *select* content, which is explained in Real-World Data: iTunes. The other kind of extraction produces a reduced database; it is dubbed *projection*. More specifically, a projection constructs a database by retaining only certain columns from a given database.

The description of a projection suggests the following:

```
; DB [List-of Label] -> DB
; retains a column from db if its label is in labels
(define (project db labels) (make-db '() '()))
```

Given the complexity of a projection, it is best to work through an example first. Say we wish to eliminate the age column from the database on the left in figure 137. Here is what this transformation looks like in terms of tables:

the original database			... eliminating the "Age" column	
Name	Age	Present	Name	Present
String	Integer	Boolean	String	Boolean
"Alice"	35	#true	"Alice"	#true
"Bob"	25	#false	"Bob"	#false
"Carol"	30	#true	"Carol"	#true
"Dave"	32	#false	"Dave"	#false

A natural way to articulate the example as a test reuses figure 138:

```
(define projected-content
  `(("Alice" #true)
    ("Bob"   #false)
    ("Carol" #true)
    ("Dave"  #false)))

(define projected-schema
  `(("Name" ,string?) ("Present" ,boolean?)))

(define projected-db
```

```
  (make-db projected-schema projected-content))
;  Stop! Read this test carefully. What's wrong?
(check-expect (project school-db '("Name" "Present"))
              projected-db)
```

```
(define (project db labels)
  (local ((define schema  (db-schema db))
          (define content (db-content db))
          ; Spec -> Boolean
          ; does this spec belong to the new schema
          (define (keep? c) ...))
          ; Row -> Row
          ; retains those columns whose name is in labels
          (define (row-project row) ...))
    (make-db (filter keep? schema)
             (map row-project content))))
```

Figure 140: A template for `project`

If you run the above code in DrRacket, you get the error message

```
first argument of equality cannot be a function
```

before DrRacket can even figure out whether the test succeeds. Recall from Functions Are Values that functions are infinitely large objects and it is impossible to ensure that two functions always produce the same result when applied to the same arguments. We therefore weaken the test case:

```
(check-expect
  (db-content (project school-db '("Name" "Present")))
  projected-content)
```

For the template, we again reuse existing abstractions; see figure 140. The `local` expression defines two functions: one for use with `filter` for narrowing down the schema of the given database and the other for use with `map` for thinning out the content. In addition, the function again extracts and names the schema and the content from the given database.

Before we turn to the wish list, let's step back and study the decision to go with two reuses of existing abstraction. The signature tells us that the function consumes a structure and produces an element of DB, so

```
(local ((define schema (db-schema db))
        (define content (db-content db)))
  (make-db ... schema ...
           ... content ...))
```

is clearly called for. It is also straightforward to see that the new schema is created from the old schema and the new content from the old content. Furthermore, the purpose statement of `project` calls for the retention of only those labels that are mentioned in the second argument. Hence, the `filter` function correctly narrows down the given `schema`. In contrast, the rows per se stay except that each of them loses some cells. Thus, `map` is the proper way of processing `content`.

Now we can turn to the design of the two auxiliary functions. The design of `keep?` is straightforward. Here is the complete definition:

```
; Spec -> Boolean
; does this spec belong to the new schema
(define (keep? c)
  (member? (first c) labels))
```

The function is applied to a Spec, which combines a Label and a Predicate in a list. If the former belongs to `labels`, the given Spec is kept.

For the design of `row-project`, the goal is to keep those Cells in each Row of `content` whose name is a member of the given `labels`. Let's work through the above example. The four rows are:

```
(list "Alice" 35 #true)
(list "Bob"   25 #false)
(list "Carol" 30 #true)
(list "Dave"  32 #false)
```

Each of these rows is as long as `school-schema`:

```
(list "Name" "Age" "Present")
```

The names in the schema determine the names of the cells in the given rows. Hence, `row-project` must keep the first and third cell of each row because it is their names that are in the given `labels`.

Since Row is defined recursively, this matching process between the content of the Cells and their names calls for a recursive helper function that `row-project` can apply to the content and the labels of the cells. Let's specify it as a wish:

```
; Row [List-of Label] -> Row
; retains those cells whose corresponding element
; in names is also in labels
(define (row-filter row names) '())
```

Using this wish, `row-project` is a one-liner:

```
(define (row-project row)
  (row-filter row (map first schema)))
```

The `map` expression extracts the names of the cells, and those names are handed to `row-filter` to extract the matching cells.

Exercise 405. Design the function `row-filter`. Construct examples for `row-filter` from the examples for `project`.

Assumption The given database passes an integrity check, meaning each row is as long as the schema and thus its list of names.

Figure 141 puts all the pieces together. The function `project` has the suffix `.v1` because it calls for some improvement. The following exercises ask you to implement some of those.

```
(define (project.v1 db labels)
  (local ((define schema  (db-schema db))
          (define content (db-content db))

          ; Spec -> Boolean
          ; does this column belong to the new schema
          (define (keep? c)
            (member? (first c) labels))

          ; Row -> Row
          ; retains those columns whose name is in labels
          (define (row-project row)
            (row-filter row (map first schema)))

          ; Row [List-of Label] -> Row
          ; retains those cells whose name is in labels
          (define (row-filter row names)
            (cond
              [(empty? names) '()]
              [else
               (if (member? (first names) labels)
                   (cons (first row)
                     (row-filter (rest row) (rest names)))
                   (row-filter (rest row) (rest names)))])))
    (make-db (filter keep? schema)
             (map row-project content))))
```

Figure 141: Database projection

Exercise 406. The `row-project` function recomputes the labels for every row of the database's content. Does the result differ from function call to function call? If not, hoist the expression.

Exercise 407. Redesign `row-filter` using `foldr`. Once you have done so, you may merge `row-project` and `row-filter` into a single function. **Hint** The `foldr` function in ISL+ may consume two lists and process them in parallel.

The final observation is that `row-project` checks the membership of a label in `labels` for every single cell. For the cells of the same column in different rows, the result is going to be the same. Hence it also makes sense to hoist this computation out of the function.

This form of hoisting is somewhat more difficult than plain expression hoisting. We basically wish to pre-compute the result of

```
(member? label labels)
```

for all rows and pass the results into the function instead of the list of labels. That is, we replace the list of labels with a list of Booleans that indicate whether the cell in the corresponding position is to be kept. Fortunately, computing those Booleans is just another application of `keep?` on the schema:

```
(map keep? schema)
```

Instead of keeping some Specs from the given `schema` and throwing away the others, this expression just collects the decisions.

```
(define (project db labels)
  (local ((define schema  (db-schema db))
          (define content (db-content db))

          ; Spec -> Boolean
          ; does this column belong to the new schema
          (define (keep? c)
            (member? (first c) labels))

          ; Row -> Row
          ; retains those columns whose name is in labels
          (define (row-project row)
            (foldr (lambda (cell m c) (if m (cons cell c) c))
                   '()
                   row
                   mask))
          (define mask (map keep? schema)))
    (make-db (filter keep? schema)
             (map row-project content))))
```

Figure 142: Database projection

Figure 142 shows the final version of `project` and integrates the solutions for the preceding exercises. It also uses `local` to extract and name `schema` and `content`, plus `keep?` for checking whether the label in some Spec is worth keeping. The remaining two definitions introduce `mask`, which stands for the list of Booleans discussed above, and the revised version of `row-project`. The latter uses `foldr` to process the given `row` and `mask` in parallel.

Compare this revised definition of `project` with `project.v1` in figure 141. The final definition is both simpler and faster than the original version. Systematic design combined with careful revisions pays off; test suites ensure that revisions don't mess up the functionality of the program.

Exercise 408. Design the function `select`. It consumes a database, a list of labels, and a predicate on rows. The result is a list of rows that satisfy the given predicate, projected down to the given set of labels.

Exercise 409. Design `reorder`. The function consumes a database `db` and list `lol` of Labels. It produces a database like `db` but with its columns reordered according to `lol`. **Hint** Read up on `list-ref`.

At first assume that `lol` consists exactly of the labels of `db`'s columns. Once you have completed the design, study what has to be changed if `lol` contains fewer labels than there are columns and strings that are not labels of a column in `db`.

Exercise 410. Design the function `db-union`, which consumes two databases with the exact same schema and produces a new database with this schema and the joint content of both. The function must eliminate rows with the exact same content.

Assume that the schemas agree on the predicates for each column.

Exercise 411. Design `join`, a function that consumes two databases: `db-1` and `db-2`. The schema of `db-2` starts with the exact same Spec that the schema of `db-1` ends in. The function creates a database from `db-1` by replacing the last cell in each row with the *translation* of the cell in `db-2`.

Here is an example. Take the databases in figure 137. The two satisfy the assumption of these exercises, that is, the last Spec in the schema of the first is equal to the first Spec of the second. Hence it is possible to join them:

Name	Age	Description
String	Integer	String
"Alice"	35	"presence"
"Bob"	25	"absence"
"Carol"	30	"presence"
"Dave"	32	"absence"

Its translation maps `#true` to `"presence"` and `#false` to `"absence"`.

Hints (1) In general, the second database may "translate" a cell to a row of values, not just one value. Modify the example by adding additional terms to the row for `"presence"` and `"absence"`.

(2) It may also "translate" a cell to several rows, in which case the process adds several rows to the new database. Here is a second example, a slightly different pair of databases from those in figure 137:

Name	Age	Present
String	Integer	Boolean
"Alice"	35	#true
"Bob"	25	#false
"Carol"	30	#true
"Dave"	32	#false

Present	Description
Boolean	String
#true	"presence"
#true	"here"
#false	"absence"
#false	"there"

Joining the left database with the one on the right yields a database with eight rows:

Name	Age	Description
String	Integer	String
"Alice"	35	"presence"
"Alice"	35	"here"
"Bob"	25	"absence"
"Bob"	25	"there"
"Carol"	30	"presence"
"Carol"	30	"here"
"Dave"	32	"absence"
"Dave"	32	"there"

(3) Use iterative refinement to solve the problem. For the first iteration, assume that a "translation" finds only one row per cell. For the second one, drop the assumption.

Note on Assumptions This exercise and the entire section mostly rely on informally stated assumptions about the given databases. Here, the design of `join` assumes that "the schema of `db-2` starts with the exact same Spec that the schema of `db-1` ends in." In reality, database functions must be checked functions in the spirit of Input Errors. Designing `checked-join` would be impossible for you, however. A comparison of the last Spec in the schema of `db-1` with the first one in `db-2` calls for a comparison of functions. For practical solutions, see a text on databases.

24 Summary

This fourth part of the book is about the design of functions that process data whose description requires many intertwined definitions. These forms of data show up everywhere in the real world, from your computer's local file system to the world wide web and geometric shapes used in animated movies. After working through this part of the book carefully, you know that the design recipe scales to these forms of data, too:

1. When the description of program data calls for several mutually referential data definitions, the design recipe calls for the simultaneous development of templates, one per data definition. If a data definition `A` refers to a data definition `B`, then the template `function-for-A` refers to `function-for-B` in the exact same place and manner. Otherwise the design recipes work as before, function for function.

2. When a function has to process two types of complex data, you need to distinguish three cases. First, the function may deal with one of the arguments as if it were atomic. Second, the two arguments are expected to have the exact same structure, and the function traverses them in a completely parallel manner. Third, the function may have to deal with all possible combinations separately. In this case, you make a two-dimensional table that along one dimension enumerates all kinds of data from one data definition and along the other one deals with the second kind of data. Finally you use the table's cells to formulate conditions and answers for the various cases.

   This part of the book deals with functions on two complex arguments. If you ever encounter one of those rare cases where a function receives three complex pieces of data, you know you need (to imagine) a three-dimensional table.

You have now seen all forms of structural data that you are likely to encounter over the course of your career, though the details will differ. If you are ever stuck, remember the design recipe; it will get you started.